SOME EXTENSIONS TO IRVINE DATA FLOW LANGUAGE (Id)

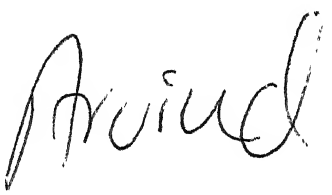
A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

By
VINOD KUMAR KATHAIL

to the
COMPUTER SCIENCE PROGRAMME
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR
JULY, 1978

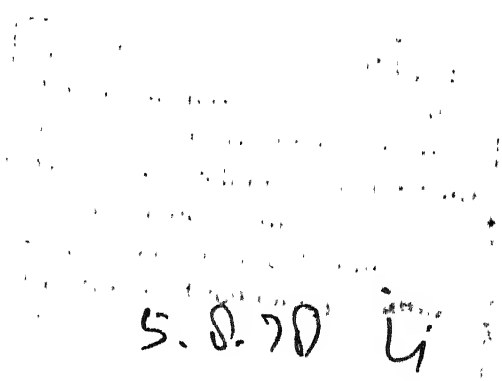
CERTIFICATE

This is to certify that the thesis entitled "SOME EXTENSIONS TO IRVINE DATA FLOW LANGUAGE (Id)" by Vinod Kumar Kathail has been carried out under my supervision and has not been submitted elsewhere for a degree.



Kanpur
July 21,1978

Arvind
Visiting Assistant Professor
Computer Science Programme
Indian Institute of Technology, Kanpur
and
Assistant Professor
Information and Computer Science
University of California, Irvine



ACKNOWLEDGEMENTS

With a deep sense of gratitude I wish to acknowledge my gratefulness to Dr. Arvind for generating and stimulating my interest in data flow languages; and for his inspiring guidance, keen interest and constructive criticisms which were invaluable towards the completion of this thesis.

Special thanks are due to Mr. V.M. Malhotra for the long hours he spent in reading the manuscript and for providing many useful suggestions. I would also like to thank my friends Mr. Vijoy Kumar, Mr. Sunil Khanna, Mr. Surendra Singh and Mr. A.K. Modi for their help during the final phase of the preparation of the thesis.

Finally, thanks are due to Mr. J.S. Rawat for careful typing of the manuscript.

Kanpur
July 21, 1978

- Vinod Kathail

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Data flow languages are gaining importance because of their inherent asynchronous view of computation. Some extensions to the high-level data flow language Id (Irvine data flow language) are proposed to increase its effectiveness and expressive power. The proposed extensions are in accordance with the recent advances in the area of programming language design and programming methodology. Sequential and mixed looping constructs are proposed which alongwith the existing parallel looping construct and with the proposed incorporation of generators provide a comprehensive and flexible set of looping constructs. Stream structures are proposed to model the arrays of streams. Their usefulness in the resource management to maintain a variable number of queues is discussed. To aid in program abstraction and to enable a programmer to define his own language, tailored to his needs, programmer defined constructs (pdcs) are incorporated. The approach used to incorporate programmer defined constructs has some similarities with the well known conditional assembly of macros used in low level languages. Usefulness and implementation of some of these aspects in conventional languages is briefly discussed.

CHAPTER 1

INTRODUCTION

1.1 Objective

Dataflow languages are gaining importance because of their inherent asynchronous view of computation and thus their ability to act as a suitable semantic basis for parallel computation. The objective of this thesis is to propose some extensions to a higher-level dataflow language Id [AGP78] (Irvine dataflow language) developed by the Dataflow Architecture Group at University of California, Irvine. Proposed extensions enhance the capabilities of Id looping constructs, provide a general model to maintain a large and variable number of queues to deal with the problems of resource management, and enable a programmer to define his own higher level constructs using available Id schemas. Our proposed extensions should be viewed in the context of the recent advances in programming language design and programming methodology some of which are briefly discussed in Section 1.3. A reader who is familiar with the dataflow languages, their advantages over conventional languages, and Id may skip rest of this chapter and the next chapter.

1.2 Conventional languages vs dataflow languages and dataflow concepts

With the speed of semiconductor devices reaching their limits due to electrical propagation delays, parallel computation has emerged as the only basis to speed up computation to solve large and complex problems. Many computer systems like ILLIAC IV, CDC STAR-100, CRAYI etc. have been built todate to exploit the power of parallel computations. However, they are still based on conventional Von Neumann architecture* characterized by sequential control and memory cells, and thus require either complex software or hardware mechanisms to bring out the parallelism in the program. A new dimension is added to the problem by the recent advances in LSI technology. Large numbers of inexpensive processors are now available and thus "distributed processing" among many distinct processors with essentially autonomous control is gaining favour. However, it is now widely recognized that under the constraints of conventional architecture and languages it is extremely difficult to efficiently utilize a large number of processors, since sequential control inhibits asynchronous behaviour and existence of memory potentially requires proper synchronization of accesses to each memory cell [Den 73, GIMT 74, Wen 75, SM 77, AG 77b, AG 77c etc.]. In past many control primitives for parallel computation (CALL and WAIT

*Conventional architecture and conventional languages are used interchangably as languages reflects the characteristics of architecture.

in PL/I, Cobegin-coend etc.) and synchronization schemes (Semaphores[Dij 68] etc.) have been proposed as extensions to conventional languages. However, program written using these constructs are limited in the degree of parallelism they exhibit. Thus to exploit the capabilities provided by recent technological advances, we need an asynchronous view of computation.

Dataflow languages have been proposed to give such a view of computation[Den 73, GIMT 74, AG 77b, AGP 78 etc.].They are based on the observation that an operation should be executed as soon as its input operands are available. The attractiveness of dataflow schemas as the semantic basis of parallel computation lies in two properties:

1. an instruction executes when and only when all operands needed for that instruction become available;
2. instructions at whatever level they might exist are purely functional in nature and produce no side effects i.e. there is no memory in conventional sense [AG 77b, Den 73].

Thus dataflow program is a set of partially ordered operations on operand values where the partial ordering is determined solely and explicitly by the need for intermediate results. Dataflow languages offer several advantages [Den 73, KOS 73, AG 77b, AGP 78] over conventional programming

languages such as absence of variables, highly modular program structure and inherently functional nature. Thus they are well suited to describe and implement asynchronous behaviour required by many complex systems such as operating systems. Several dataflow languages and schemas [Den 73, KOS 73, Bah 72, KM 66 etc.] have been proposed. In particular dataflow language proposed by Dennis [Den 73] is well suited as a base language for a dataflow machine. Recent efforts have been in the direction of developing higher level languages so that dataflow becomes an attractive alternative for the end users. Weng [Wen 75] proposed a Textual Dataflow Language which incorporates recursive schemas and streams. Kasinski [Kos 73] and Bryant [Bry 77] have also proposed languages based on dataflow concepts. Recently Hankin etal [HOS 78] have proposed a language CAJOLE which is still in a developmental stage. However, it allows the programmer to define his own new constructs. Dataflow Architecture group at University of California, Irvine has proposed a higher level dataflow language Id and a computer architecture suitable for its execution [AGP 78, AG 78]. Id is of a more advanced nature and incorporates ideas of streams, monitors, nondeterministic programming, abstract data types and operator extensiality. In this thesis we will look at some of these features of Id more carefully.

1.3 Programming languages in general

In recent years production of reliable software has drawn a lot of attention. It has been noted that conventional languages are not well suited for program verification and for writing reliable complex software systems. Structured programming has been proposed to bring order and structure in the otherwise chaotic world of programming languages [Bac 78]. It is now well understood that development and maintenance of complex software systems is facilitated by the preservance of the problem structure in its solution and by modularization, to localize the subsequent modifications. Drive is towards a functional view of computation rather than a procedural view. Current interest in data abstraction and control abstraction [LSAS 77, SWL 77] points in this direction. To aid the verification of programs, the structure of looping constructs is being studied. It has been proposed that initialization of loop variables, loop terminating conditions and incrementation of loop variables should be explicitly stated and grouped to aid in program verification [Pra 78]. Languages whose semantics is more closely akin to mathematics have been proposed to simplify program verification [Gut 77, AW 77].

Another area which is being extensively explored, concerns with the linguistic aspects of the problems related

to resource management [JL 76]. A higher level language especially a language suitable for distributed processing should have capabilities to handle hardware and software resources. Here also we see a drive towards more encapsulated and functional approach (semaphores [Dij 68] to monitors [Hoa 74]).

With the Backus' proposal [Bac 78] a new school of thought is emerging in the field of programming language design. His main objection against the conventional programming languages is that they are growing ever more enoromous, but not stronger because of their inherent defects at the most basic level. They are founded on word - at - atime programming inherited from their ancestor Von Neumann computer and their semantics is closely coupled to the state transition. They have more rigid parts and less variable parts because of their closed coupling to the state transition and dividing the programming into a world of statements and world of expressions. Whereas, the world of expressions is orderly and has useful mathematical properties, the world of statements is chaotic. Because of these aspects they are unable to effectively use powerful combining forms to build new programs from the existing ones, and they lack useful mathematical properties. His proposal is to adopt a truely variable less functional programming language which has less rigid parts and more variable

parts. Such functional programs are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments and donot require complex procedure declarations. Functional programming can easily use higher level programs to build still higher level ones. Since programs, their semantics, and their realizations are defined in the same language, a simple algebra of programs can be used for program transformations.

We believe that some of the aspects discussed above are important for the next generation of programming languages which will be required to deal with systems of still-yet-unimagined complexity.

1.4 Overview of the thesis

Since the thesis extensively uses the presently available Id constructs to describe proposed extensions, Chapter 2 briefly describes Id and its constructs. Block expressions, conditional expressions, loop expressions, procedures, monitors, and programmer defined data types are discussed. To show the use of some of these constructs, Id program to compute Fast Fourier Transform of a signal sequence is described.

Chapter 3 deals with sequential looping constructs, mixed looping constructs and generators. An alternate translation of for loops using for each loop and generator is

discussed. This view of for loops makes the translation of mixed for loop constructs easy. A simplified translation of for each - while loop is also discussed, and, its usefulness in the controlled generation of the elements in a for-while loop and in a pdt, is shown.

Chapter 4 describes stream structures which are useful in maintaining a variable and a large number of queues. Dynamic stream construct of Id is reviewed and some base language operators are extended to allow streams as creation time parameters for Id monitors. Usefulness of the stream structures in the resource management problems is shown through an example.

Chapter 5 discusses the programmer defined constructs, which enable a programmer to define his own constructs and aids in program abstraction. Syntax for its definition and its use, and, its implementation is described. To bring out some of the relevant points and limitations several examples are discussed.

In Chapter 6 we present our conclusions and observe that some of the concepts discussed in the thesis can also be incorporated in conventional languages to enhance their capabilities.

CHAPTER 2

Id: A BRIEF INTRODUCTION

This chapter is intended to make the reader familiar with the high level dataflow language Id (Irvine dataflow) together with the syntax and . the underlying semantics of its constructs [AGP77, AGP78].

2.1 Id and the underlying model of execution*

Id is a block structured, expression oriented, single assignment language whose syntax resembles Algol-type of block structured language. Id provides facilities for resource handling via dataflow monitors; incorporates data abstraction mechanisms (i.e. programmer defined data types), and is extensible.

Id like any other dataflow language is based on the principle of dividing computation into smaller subcomputations called activities which may be executed by independent processing elements. Every Id program is compiled into a corresponding program in the base language which is an ordered graph consisting of operators interconnected by directed lines that transport values in the form of tokens. A variable in

---

*This section or rather this complete chapter relies heavily on the description of Id given in [AGP78] .

an Id program does not represent the name of a memory cell, rather it is the name of a line in the base language graph along which the value travels. Id variables are not typed. The internal representation of values is self identifying and thus, type is associated with a value and not a variable. A given variable in Id or a line in program graph is either a simple variable or a stream variable [Wen 75]. A simple variable carries a single token to each instance of an operator's execution, while a stream variable carries a linearly ordered sequence of tokens to each instance of an operator's execution. Id variables are strongly typed as far as streams are concerned. Throughout the thesis simple variables will be denoted by lower case letters and stream variables by upper case letters. Base language programs corresponding to Id programs are executed on a dataflow machine using Unravelling Interpreter [AG77a]. The unravelling interpreter dynamically removes ordering between operations due only to time, while retaining orderings required by the need for partial results. This scheme of interpretation does an automatic unfolding of loops and permits simultaneous execution of distinct invocations of the same operation. This brings out far more asynchrony then what is normally possible in a dataflow language [Den 73, AG77a]. Underlying dataflow machine does not have memory in the convntional sense. The values are carried by tokens.

Memory is used only to store values that are too large to be moved efficiently; instead a pointer to stored value is carried by the token. Thus existence of memory is completely transparent to the Id programmer.

2.2 Id values and basic schemas

Id values are of nine types: integer, real, boolean, string, structure, procedure definition, monitor definition, monitor object, and errors. First four need no explanation. We will explain structures here, rest of them will be treated in later sections. A structure value is either the empty structure Λ or a set of < selector: value > ordered pairs. A selector is an integer, real, boolean, or string value; a value is any Id value. There are exactly two operators defined over structures -- SELECT and APPEND [AGP78]. The important thing to note is that an append operation always creates a logically new structure, and the original structure may enjoy a simultaneous existence. Multiple selectors are allowed.

We explain three basic schemas in Id: block expression conditionals, and loops.

2.2.1 Block expressions:

Any Id program can be written as a list of expressions. However, it is convenient to break a computation and identify certain partial results. An example of block expression is

```
x,y ← (x ← b * 2 + a;
       y ← 2 * a
       return x + y,y)                    (2.1)
```

The inputs to the block expression are those variables which are referenced but not assigned in the block and outputs (ordered) are specified by return clause. The compilation of expression is shown in Figure 2.1. An assignment statement names the line in corresponding base language graph. Id limits the scope of names to the block in which they are defined. Thus x,y have different values inside and outside the block in expression (2.1). A constant like 2 in Id does not represent a value but function that produces that value as its output regardless of the value of the input. Box C in Figure 2.1 represents such a function.

2.2.2 Conditional expressions:

A conditional expression in Id is of the form

```
(if p(x) then f(x) else g(x))             (2.2)
```

Its base language translation is given in Figure 2.2. The SWITCH operator passes the token coming on data line to the T branch or the F branch depending on whether token on control line carries the value true or false. If any of the branch (T or F) is not connected and the control input dictates that branch to be taken, then the data token is simply absorbed. The

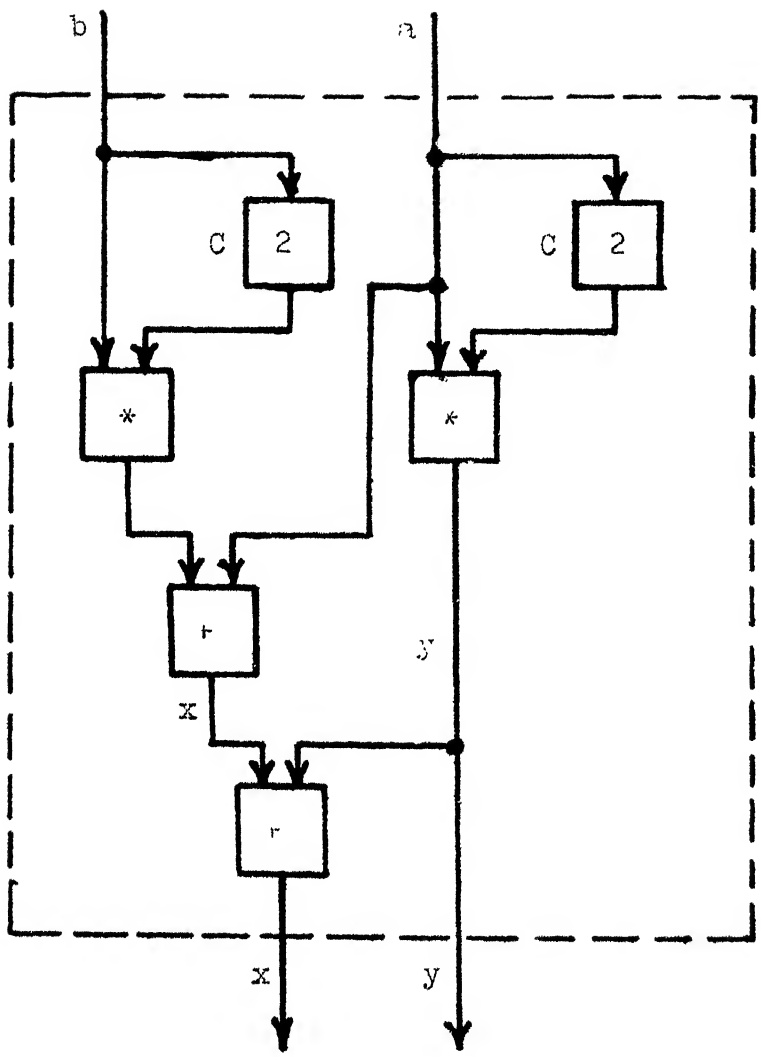


Figure 2.1 Compilation of the block expression (2.1).

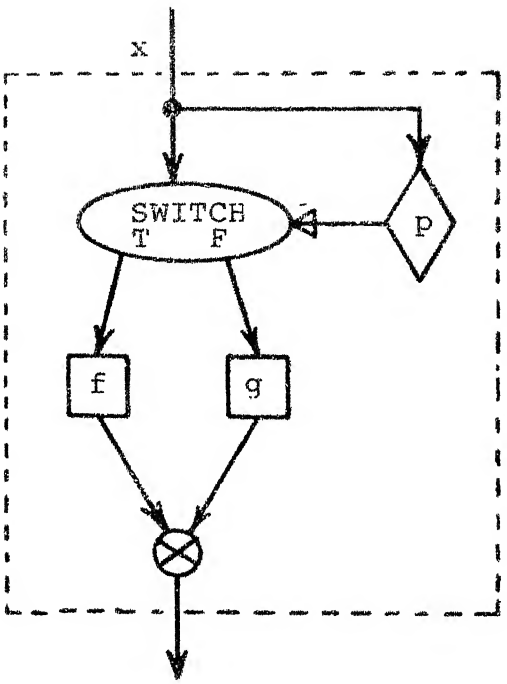


Figure 2.2 Compilation of the conditional expression (2.2).

Ⓧ operator merges the two branches of conditional expression. A conditional expression needs all of its inputs for execution regardless of the branch to be taken. In a conditional expression the then clause and the else clause should contain equal number of expressions. Conditionals can also be written in the statement form. Thus expression (2.3) and expression (2.4) are equivalent.

(if p(x) then y ← f(x); z ← 1 else y ← g(x); z ← 0) (2.3)

y,z ← (if p(x) then f(x), 1 else g(x), 0) (2.4)

2.2.3 Loop expressions:

Basic loop construct in Id is a while loop (loops involving streams are different). A while loop is given in expression (2.5) and its compilation in Figure 2.3

```
(initial i ← 1; sum ← 1
 while sum ≤ s do
   new i ← i + 1;
   new sum ← sum + i
 return  sum, i)                    (2.5)
```

Variables initialized in the initial part and all loop constants such as s circulate in the loop domain. For the next iteration of the loop their new values are defined by loop body. A default for loop constants is that their new values

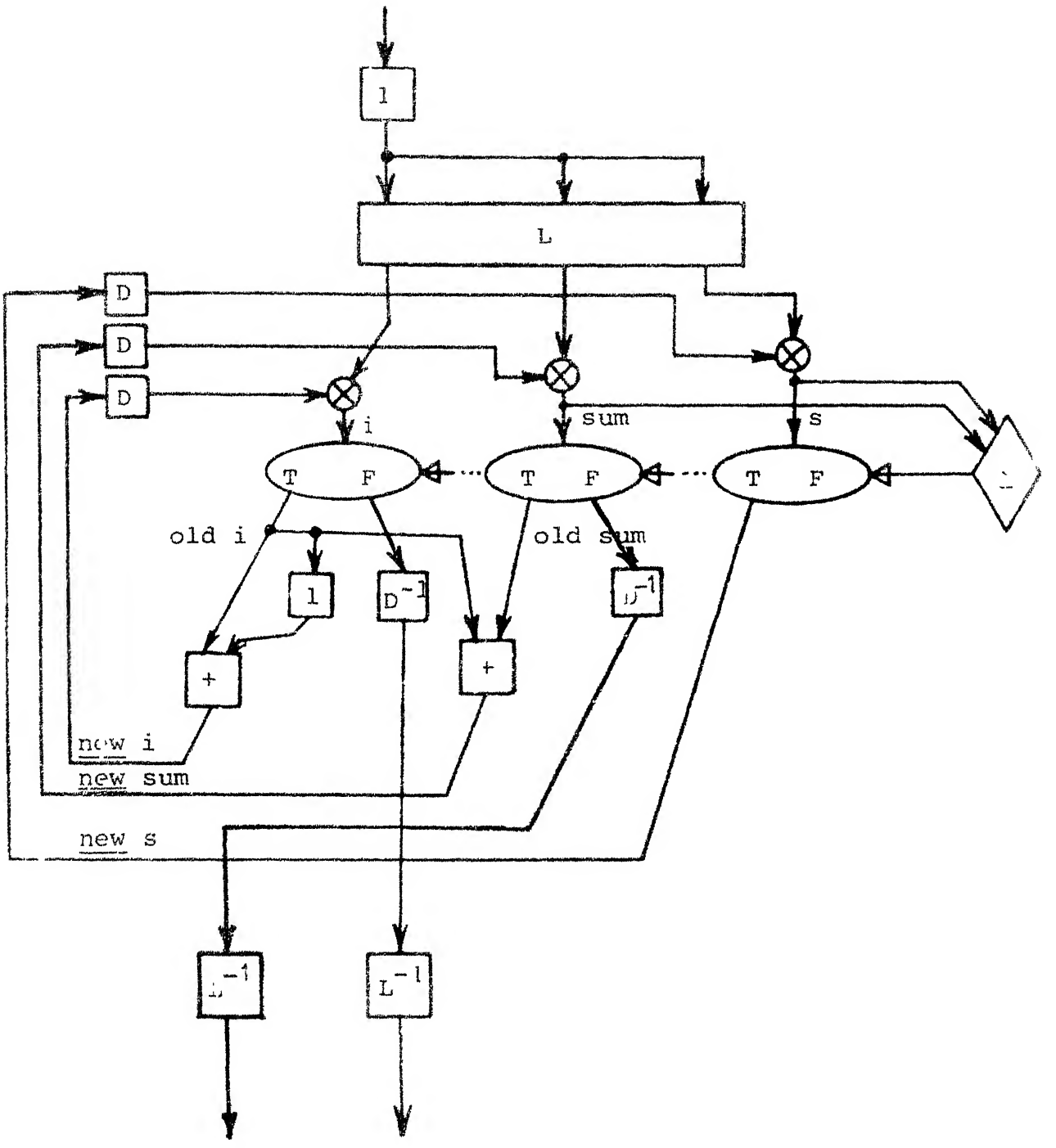


Figure 2.3 Correlation of the loop expression (2.1).

are same as old values. The operators L, L^{-1}, D, D^{-1} do not affect the values of the tokens passing through them. They are used by the unravelling interpreter to manipulate activity names so that tokens belonging to different instantiations of a loop do not get mixed up with the tokens belonging to different iterations of a loop instantiation. new i is used to label a line which is different from i and does not violate the single assignment rule.

Id supports many kinds of looping constructs such as for loops, for-while loops etc. However, they are compiled in form of a basic while loop. Expression (2.6) gives a for-while loop whose equivalent while loop is given in expression (2.7).

```
(initial  x ← f(x)
 for i from  1 to n  while p(x) do
   y ← g(i);
   new x ← f(x)+y
 return x)                                   (2.6)

(initial  x ← f(x)
 while i≤ n ∧ p(x) do
   y ← g(i);
   new x ← f(x) + y;
   new i ← i+1
 return x)                                   (2.7)
```

A basic looping construct involving streams is given in expression (2.8).

```
(initial   s ← a
 for each x in X; y in Y while p(x,y) do
   new s ← s+x+y;
   z ← f(x,y);
   if z≤ t then b ← next p
          else b ← λ
 return s, all z, all b but λ)                    (2.8)
```

There are many points brought out by this expression which are relevant when dealing with streams. A *for each* loop operates on all the elements of a stream. Expression (2.8) shows a parallel looping construct in which for each iteration of a loop an element from X and an element from Y are taken. Loop terminates when predicate p is satisfied, or either of the two streams runs out of tokens. *all* z is a stream producing construct and returns a stream containing all the values of z. *but* λ performs what one intuitively expects it to do: it filters out λ's from the stream *all* b. *next* P construct gives next element from the stream P, when asked, and thus avoids circulating the streamP. *next* only has meaning when it is inside a conditional construct which is embedded in a loop expression. Compilation of expression (2.8)

is too involved and is given in Sections 4.2.3.2 and 4.2.3.3 of [AGP78]. We will show a simpler implementation of the loops involving next in Chapter 4.

2.3 Procedure definition and application

A procedure definition in Id is a constant value and it may be created by writing:

```
p ← procedure (x) (if x > 1 then x - 1
                           else x + 1)      (2.9)
```

x is a formal parameter, and the expression is the procedure body. p is the name of the line along which the token carrying the procedure value will travel. In Id two operators are defined over procedure values: APPLY and COMPOSE. Apply applies the procedure definition to a list of actual arguments. We write an application as

```
res ← apply (p, arg )
```

where arg is actual argument. A syntactic shorthand is

```
res ← p (arg)
```

The primitive apply is implemented as two base language operators: A and A^{-1} as shown in Figure 2.4. The A actor accepts the procedure definition and creates an instance of its execution. It then sends the actual

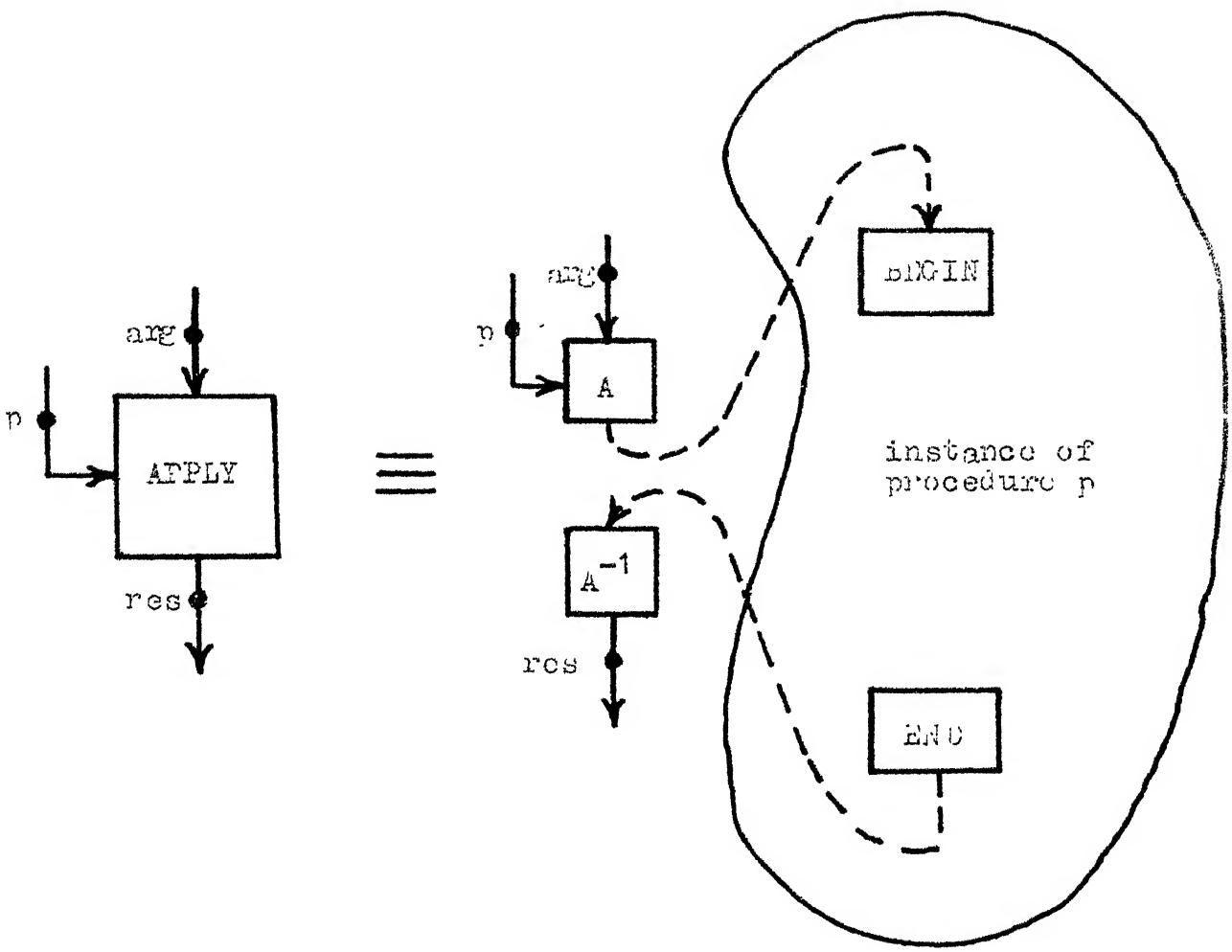


Figure 2.4 A procedure application.

arguments to this instance. The results of computation are returned to A^{-1} and the instance of the procedure is automatically destroyed. Distinct instances are created for the application of the same procedure at different places.

Compose operator takes the input procedure value and "freezes" one or more of the procedure's formal parameters to particular actual values. For example, in

```
a ← procedure (b,C) (b * c + b/c);
d ← compose < a,  < 2 > >                    (2.10)
```

d behaves as if programmer had written

```
d ← procedure (c) (2 * c + 2/c)
```

It is possible to give a name to an Id procedure. This is useful in writing recursive procedures. Whenever a named procedure is applied its definition is also passed as an argument. Expression (2.11) gives a recursive procedure to calculate factorial function.

```
y ← procedure f(n) (if n = 0 then 1 else n * f(n-1))
                                             (2.11)
```

compose and named procedures are useful in programmer defined data types. Procedure's arguments are not limited to simple variables, stream variables are also allowed.

2.4 Dataflow monitors*

The concept of a dataflow monitor [AGP77, AGP78] was introduced in order to allow nondeterministic behaviour generally encountered in resource management. Expression (2.12) assigns a dataflow monitor definition value to variable md.

```
md ← monitor  (s_o)
    (entry REQ do
       RES ← (initial s ← s_o
              for each req in REQ do
                new s, ans ← f(s, req)
              return all ans
     exit RES)                                   (2.12)
```

Many instances of monitor objects may be created from this monitor definition. Creation of a monitor instance is done by

```
me ← create  (md,a)                              (2.13)
```

A programmer refers to the monitor object by writing me. a is the initial state passed at the creation time. An

*In [AGP78] word manager is used in place of monitor.

instance of a monitor is not created and destroyed for only one invocation, but may be reused arbitrarily during its life span by sending to it arguments, which will be processed in a FIFO order. There exist only one instance of me and all the uses of me refers to that instance only.

A monitor instance is shown in Figure 2.5. The entry operator receives all requests and forms a stream by merging the requests nondeterministically. This stream is passed to the monitor's body. The exit operator converts the output stream to simple tokens and returns them to the place from where requests originated.

To use a monitor object we write

$$z \leftarrow \underline{use}\ (me,\ y) \qquad (2.14)$$

The compilation of this statement is given in Figure 2.6. The U operator sends the argument (y) to the domain indicated by me. The result is returned to U^{-1} operator and assigned to variable z.

Since the internal state of a monitor acts as a memory, the processing of a particular request may depend on the history of the previous inputs sent to the monitor.

Id provides another nondeterministic operator: merge. It merges the streams specified in its argument list

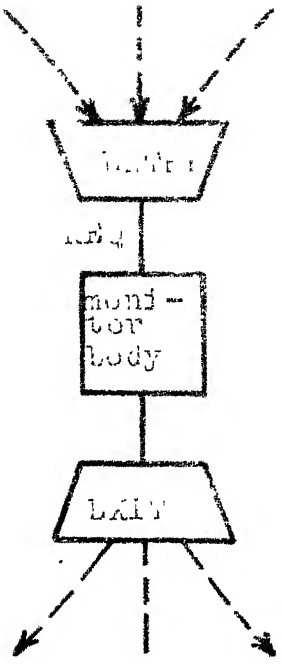
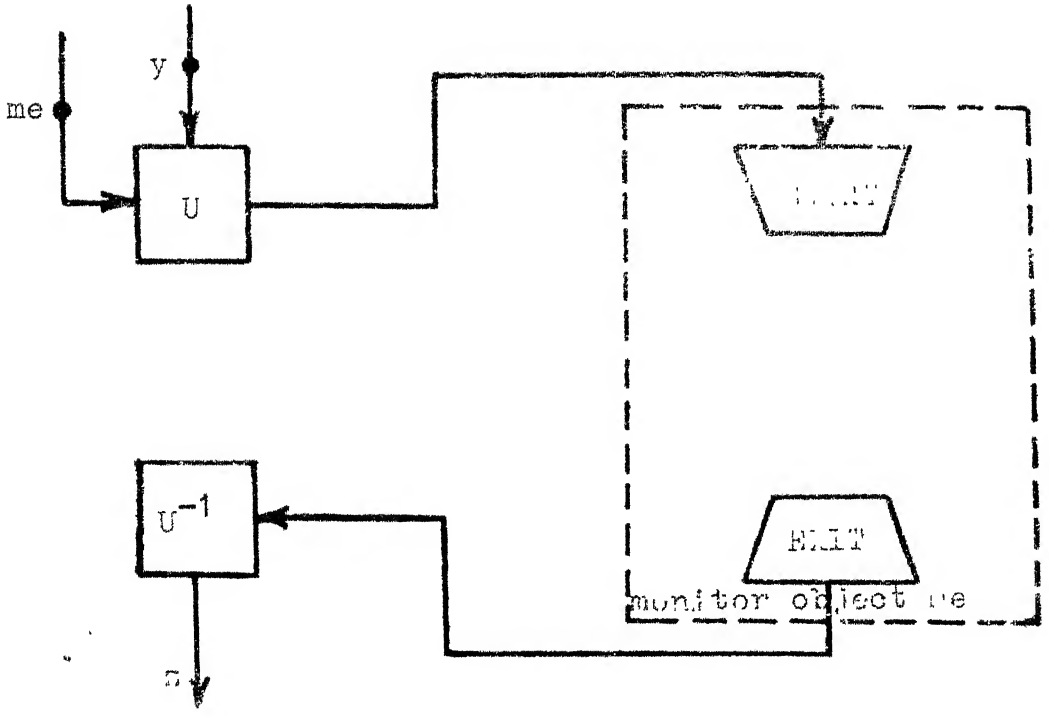


Figure 2.5 An instance of a monitor.



nondeterministically and creates a new stream. Expression (2.15) shows the use of merge.

C ← merge (A,B) (2.15)

2.5 Programmer defined data types

In Id, a procedure definition value is treated exactly like any other value. This general view of a procedure allows Id to provide some very sophisticated higher-level concepts while requiring only minimally expended lower-level mechanisms. These higher-level concepts include programmer-defined data types (pdts) and operator extensionality so operators such as "+" may be interpreted at execution time in the context of the types of their arguments.

Expression (2.16) gives a pdt 'set' with four operations. Operation 'ε' decides set membership and returns a boolean value. '+' denotes set insertion and returns a new set value. New set value is created using the procedure set-gen. 'U' and '$\cap$' denote set union and set intersection operations.

```
procedure set-gen (s,l)
(y← procedure set (! function! f, ! arguments! u,v,
                   ! set array! s, !cardinality! l, set-gen)
    (if f= "ε" then (initial t← false;
                     for i from 1 to l while t do
                       new t ← s[i] = v
                     return t)
     elseif f = "+" then (if |ε| (u,v) then u
                                       else set-gen(s+[l+1]v,
                                                    l+1))
     elseif f = "U" then (initial b ← v
                          for i from 1 to l do
                            new b ←b+ s [i]
                          return b)
     elseif f =" "then (initial b ← set-gen ( )
                        for i from 1 to l do
                          (if|ε|(v,s [ i ])then new b ← b+s[i])
                        return b)
     else error: (<'invalid set operation', f > ))
return (if s =ξ then compose (y, <,,, Λ ,0, set-gen> )
                else compose (y, <,,, s, l, set-gen>-)))
```

(2.16)

Actual values of sets are created by calling the procedure set-gen. The procedure set-gen returns a procedure value 'set' with two frozen parameters s and l which characterize that set. Thus a pdt is always a procedure. To generate a new set a we write

a ← set-gen (s,l)

where structure s contains the elements of a and l is the cardinality of a. Similarly to create an empty set we write

a ← set-gen ( )

An operation, such as set insertion (+), can be performed by writing

b ← | + | (a, 17) (2.17)

Expression (2.17) creates a new set b and it is translated as

b ← <u>apply</u> (a, +, a,17) (2.18)

Expression 2.18 can also be written as

b ← a + 17

and in this case the meaning of operator + is determined at execution time.

If the operator + receives a procedure value on line a, it dynamically changes itself to an <u>apply</u> operator to evaluate the expression of the form (2.18).

2.6 An example

In the previous sections we have discussed the mechanisms provided by Id. To show the use of these mechanisms, we give an Id program to compute the Fast Fourier Transform of a discrete signal sequence. The signal sequence is represented as a stream.

```
procedure fast-fourier (X, j, n, w)
                          !X is the stream of the signal samples,
                            n is the number of the samples in
                            the original stream, w contains the
                            the values of W_n^k , where W_n =
                            e^(-j 2π/n) , and j=2^i, where i denotes
                            the level at which we are working.
                            Initially j should be 1.!
(if empty (rest (X)) then X
 ! stream is divided into two parts containing odd and even
   numbered samples.
 else ( P,Q ← (initial flag ← true
               for each x in X do
                 (if flag then p ← x; q ← λ )
                          else p ← λ ; q ← x);
                 new flag ← ¬ flag
               return all p but λ, all q but λ );
        S ← fast-fourier (P, 2*j, n, w);
        T ← fast-fourier (Q, 2*j, n, w);
        U,V ← (initial i ← 0
               for each s in S; t in T do
                 a ← w[n, i*j ] +t ;
                 u ← s + a ;
                 v ← s - a ;
               new i ← i + 1
               return all u, all v)
        return concatenate (U,V)))                    (2.19)
```

Keeping the asynchronous input and output behaviour of the streams in mind, a rough analysis of the above program will show that it takes $O(n+\log_2 n)$ time, whereas, a corresponding program on a sequential machine will take $O(n \log_2 n)$ time [KAT78b]. However, the above program requires atleast $O(n)$ processors to execute in the time complexity of $O(n+\log_2 n)$. This proves the point that Id is able to exchange processors with time. As a passing point we would like to mention that most of the algorthims, such as merge-sort [KAT78a] etc., show this tendency. However, certain conventional algorithms which are efficient on sequential machines turn out to be inefficient when translated in Id. Thus it is reasonable to assume that to exploit asynchrony inherent in the problem it is not sufficient to translate existing algorithms into Id or any other dataflow language. Instead new algorithms must be invented and analyzed to derive benefits of asynchrony [Mal 78].

CHAPTER 3

SEQUENTIAL LOOPS, MIXED LOOPS, AND GENERATORS

In this chapter we discuss many aspects of looping constructs in Id and show that the idea of generators and iterators can be incorporated in Id very easily. As a complement to parallel loops we propose sequential looping construct, that is loops of nested nature, and generalize these constructs to include any combination of sequential and parallel loops i.e. mixed looping constructs. Sections 3.2 and 3.3 deal with sequential looping constructs and their efficiency considerations. Section 3.4 explains syntax and translation of mixed *for each* loop constructs. Section 3.5 shows how the idea of generators can easily be incorporated in Id. Section 3.6 presents an alternate translation of *for* loop in terms of *for each* loop using the concept of a special generator i *to* j. This view of *for* loops makes the translation of mixed *for* loop constructs easy. Section 3.7 discusses a simplified translation of *for each* - *while* loop. Section 3.8 extends the ideas developed in Section 3.7 to control the generation of elements in a *for* - *while* loop and in a pdt. Finally Section 3.9 shows how the ideas of sections 3.4, 3.6, 3.7 and 3.8 can be used to translate mixed looping constructs with *while* clause.

3.1 Background and motivation

As described in Chapter 2, Id [AGP 78] provides many kinds of looping constructs to operate on simple as well as stream variables. For the sake of comprehensiveness we list them below.

(a) while p(x) do where p(x) is some boolean expression
(b) for i from j to k by 1 do and
    for i from j to k by 1 while p(x) do
(c) for each x in X do
(d) for each x in X; y in Y do
(e) for each x in X; y in Y while p(x,y) do

Loop expressions like any other expressions in Id can be nested to any arbitrary depth. At present, loop expressions involving next in Id are different from (a) to (e). However, with a simpler implementation of next described in Chapter 4 (section 4.5), the present discussions will be valid to loop expressions involving next also.

Our motivation to go for sequential and mixed looping constructsis to make programmer's task easier. Consider an Id program to sum all the elements of a n x m matrix given in expression (3.1).

```
(initial sum ← 0
 for i from 1 to n do
   new sum ← sum + (initial  sum ← 0
                    for j from  1 to m do
                      new sum ← sum + a[ i,j]
                    return sum)
 return sum )                                        (3.1)
```

Id forces one to initialize the sum twice. One can imagine the programmer's burden when the levels of nesting goes beyond three or four. Syntactically it will be more convenient for a programmer to write

```
(initial  sum ← 0
 for i from 1 to n,
     j from 1 to m do
       new sum ← sum + a[ i,j ]
 return sum)                                         (3.2)
```

However, translation of above construct is rather difficult in terms of existing compilation of for loops. The translation is considerably simplified if we view 1 to n as a generator generating successive values of the loop variable i. Another motivation to incorporate generators is to extend looping constructs to iterate over the elements of a pdt without giving undue emphasis on the order of enumeration.

Hiding the order of enumeration is a useful concept because in many data structures, such as sets, there is no implicit ordering (in mathematical sense) defined over their elements. CLU [LSAS77], ALPHARD [SWL77] and EUCLID [LHLMP77] already provide such an extended view of looping constructs to iterate over the elements of an abstract data type.

3.2 A sequential looping construct for ID

Since a semicolon is used to define a parallel looping construct (see (d) and (e)) let a comma separating two clauses of a loop (as in expression (3.2)) define their sequential or nested nature*. Hence the meaning of

for each i in I, j in J do

is that for each element of i take each element of J one by one and execute one iteration of the loop. Hence the loop body will be executed |I| X |J| times as opposed to min{|I|, |J|} times (as in the case of a parallel loop). Implementation of a sequential loop is quite straightforward and is explained below in terms of an equivalent Id program not involving such a construct.

---

*The use of comma here is different from UCI dataflow Note 11 (revised).

```
(initial x ← x_o
 for each i  in I, j in J do
   new x  ←f(x,i,j)
 return x)                                   (3.3)
```

The loop in expression (3.3) can be implemented as

```
(initial x ← x_o
 for each i  in I do
   new x ←(initial ~x←x
           for each j in J do
             new x ← f(x,i,j)
           return x)
 return x)                                   (3.4)
```

There are some subtle issues involved in the correct translation of expressions contained in the return clause of expression (3.4). An equally efficient but somewhat more straightforward translation of (3.3) is given in (3.5). The expression in (3.5) does not affect the body of the loop. Only the input streams I and J are suitably enlarged to I' and J', each of which has a length $|I| \times |J|$. We prefer to translate expression (3.3) into expression (3.5) rather than expression (3.4) as the scheme of expression (3.5) is useful in translating mixed looping constructs.

```
(I', J' ← (for each i in I do
              I',J' ← (for each j in J do
                          return all i, all j)
           return all I', all J')
 return (initial . x ← x_0
         for each i in I'; j in J' do
           new x ← f(x,i,j)
         return x))                                   (3.5)
```

The construct return all X returns a stream as opposed to a stream of streams.

3.3 Efficiency of sequential loop expression

Expression (3.2) for summing all the elements of a matrix can be translated into expression (3.6) in a mechanical way.

```
(initial . sum ← 0
 for i from 1 to n do
   new sum ← (initial sum ← sum
              for j from 1 to m do
                new sum ← sum + a[i,j]
              return sum)
 return sum)                                          (3.6)
```

Note that expression (3.6) is considerably less asynchronous than expression (3.1) due to the fact that all

the additions in (3.6) are done sequentially. In expression (3.1) summation of elements within a row is still sequential but it proceeds simultaneously for all the rows. While expression (3.1) will take O(n) time, expression (3.6) will take $O(n^2)$ time. It is possible to mechanically transform expression (3.6) into expression (3.1) but this is quite a difficult task because it involves meanings of many non-control operators in Id. It is reasonable to assume that in general when nested loop expressions are expressed as sequential loops a loss of asynchrony is involved. Thus the ease in writing may come at the expense of efficiency.

3.4 Mixed looping constructs

In order to generalize parallel and sequential looping constructs we consider the following two proposals.

(i) for each (i in I; j in J), (k in K; l in L) do

(ii) for each (i in I, j in J); (k in K, l in L) do

The loop in (i) should go through min{|I| , |J|} X min{|K| , |L|} iterations while the loop in (ii) should go through min{|I| X |J|, |K| X |L|} iterations. Semantics of expressions of type (i) are given in expression (3.7) while the meaning of (ii) is given in expression (3.8).

```
(I', J', K', L' ← (for each i in I; j in J do
                      I', J', K', L' ←(for each k in K;
                                          l in L do
                                       return all i, all j,
                                              all k, all l)
                    return all I', all J', all K', all L')
return (for each i in I'; j in J'; k in K'; l in L' do
                    .
                    .
                    .))                                    (3.7)

(I', J' ← (for each i in I do
             I', J' ←(for each j in J do
                        return all i, all j)
           return all I', all J');
K', L' ← (for each k in K do
             K',L' ←(for each l in L do
                       return all k, all l)
           return all K', all L')
return (for each i in I', j in J'; k in K'; l in L' do
                    .
                    .
                    .))                                    (3.8)
```

It is clear from these examples that we can build quite complex for each constructs by combining i in I type of clauses

either in a sequential way or in a parallel way. Implementation of such complex *for each* constructs requires no additional base language operators.

3.5 Stream producers as generators

As pointed out in Section 3.1 a useful control abstraction present in CLU [LSAS77] and ALPHARD [SWL77] is that of a generator. It offers a way of hiding the order of enumeration of the elements of an abstract data type when such an order is unnecessary. For example a generator may enumerate all the elements of a set or all the elements of an array for addition without making the order of enumeration explicit. A related problem is to enumerate the elements of a programmer defined data type when its internal structure is not known.

Something equivalent of generators can be defined very easily in Id using streams. In general *any expression producing a stream* can be treated as a generator. The elements of a stream X can be enumerated using a *for each* x in X *do* construct. Since the order of enumeration depends upon the construction of X, the problem of hiding the order of enumeration becomes the problem of hiding the generation of X. Since X will be generated by an expression and the internal functioning of no expression in Id is visible from outside, the problem of hiding or abstracting is essentially solved. We elaborate this point

further by giving examples. Suppose we want to sum the elements of a set s where s is a programmer defined datatype in Id. We could accomplish this in the following way in Id.

Define a function "enumerate" on sets and include it in the procedure set-gen (see Section 2.5). The procedure set (produced by procedure set-gen) when supplied with "enumerate" as a parameter would produce a stream containing all the elements of the set. The order of enumeration will depend upon the definition of the function "enumerate" inside the procedure set. The programmer using sets would just write

for each e in |enumerate| (s) do

Since the definition of a procedure application in Id allows streams to be input as well as output there is no problem in generating a stream from a pdt. It is also clear that such enumeration procedures can be defined for any pdt. However, we can avoid generating a stream from the pdt, if we desire so, by breaking the operation into two parts. Define a function on a pdt that gives all its elements stored in a linear structure with selectors 1 to n. A procedure that converts such an array into a stream is trivial to write. This method, however, fails with those pdts that have countably infinite number of elements. We will discuss this problem further in section 3.7.

3.6 An alternate translation of for loops using a special generator

Values of i generated by the for loop

for i from j to k do

can also be modelled as the following stream

```
I ← (initial i ← j
     while i ≤ k do
       new i ← i+1
     return all i)                    (3.9)
```

The stream I can now be used in place of from j to k as

for each i in I do

Eventhough the loop for i from j to k do results in a different base language translation, the answers produced by two loop expressions will be identical except in one detail. Suppose the last value of i is to be returned from the loop. The from j to k loop will produce the value k+1 while for each i in I loop will give an error. Several solutions to this minor problem are possible and we will not discuss them here any further. We would like to point out that both for loop and for each loop in this case are equally efficient.

One might be wondering about our motivation to translate for i from 1 to n type of loops into slightly more

complicated for each i in I type of loop. As we hope to show in the next few sections this approach helps considerably in translating sequential looping constructs of the type given in expression (3.2). Without this technique of converting for loops into a for each loop implementation of the following construct is extremely difficult.

```
for (i from 1 to n,  j from 1 to m);
    (k from 1 to q,  l from 1 to r) do        (3.10)
```

Let us introduce a new stream expression of the following type in Id

```
I ← j to k.
```

The meaning of this expression is given in (3.9). The translation of the mixed looping construct of (3.10) now can be explained as follows:

```
(I ← 1 to n; J ← 1 to m; K ← 1 to q; L ← 1 to r
 return (for each (i in I, j in J); (k in K, l in L) do
                                        ...))   (3.11)
```

Similarly we can translate any type of mixed looping construct.

3.7 For each - while type of loop revisited

Consider the following type of loop expression whose translation is discussed in [AGP78] in some detail.

```
(initial : x ← x_0
 for each b in B while p (b,x) do
   new x ← f(b,x)
 return x)                                   (3.12)
```

The main feature of the translation is that a signal stream S controls the release of tokens from stream B into the loop. Or equivalently a token of stream B enters the loop on demand only. How we will describe a slight modification of the scheme given in [AGP78].

The translation of expression (3.12) can be simplified to the one given in Figure 3.1 provided the loop of expression (3.12) satisfies the following property P.

$$p(b_i, x_i) = \underline{\text{false}} \Rightarrow b_{i+1} = \underline{\text{est}}$$

where b_i is the ith token of stream B and x_i is the input value for the ith iteration of the loop. The property P guarantees that stream B runs out of tokens exactly when the loop predicate p turns false.

In general property P does not hold for a loop. However we can always convert any for each - while loop to one for which property P does hold good. The technique is explained below by using the same example (i.e. expression (3.12)).

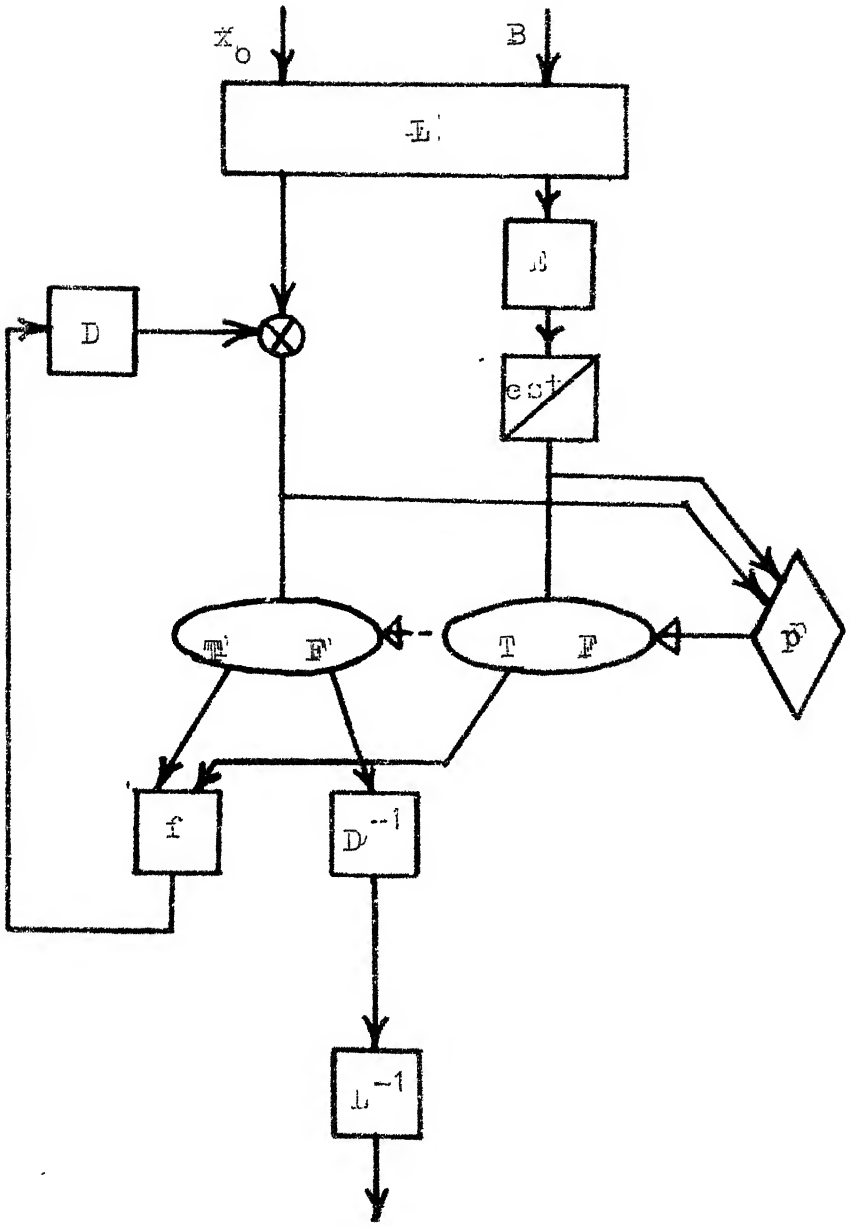


Figure 3.1 Simplified <u>for each-while</u> loop.

However, now we assume that property P is not satisfied by the loop in (3.12).

```
(x, S' ← (initial x ← x_0
          for each b in B' while p(b,x) ∧ b ≠ ε do
            new x ← f(b,x)
          return x, all true);
   S ← cons (true, S');
   B' ← exif (B,S, ε)
 return x)                                        (3.13)
```

It is clear that stream B' alongwith predicate p(b,x) $b \neq \varepsilon$ satisfies property P provided no ε tokens are present in B. This is the same condition as in [AGP78b]. The translation of expression (3.13) differs from the translation of expression (3.12) according to the rules given in [AGP78] only to the extent that exif and cons will be outside the loop domain.

3.8 Controlled generation in a for - while loop and in a pdt

We consider the translation of the following looping construct

```
for i from 1 to n while p(x) do                   (3.14)
```

An equivalent form involving generators will be

```
for each i in I while p(x) do                     (3.15)
```

where I is the stream 1 to n.

The loop in (3.14) generates a value of i only when p(x) for the previous value of i is true. Hence if n > 10 and p(x) turns false at the 10th iteration, the values 11,12, ..., n for i are never generated. On the other hand, loop in (3.15) will always get in stream I the whole sequence 1,2,..., n eventhough it will simply absorb value 11,12,.... If n happens to be a infinitely large number, the program in (3.15) will not terminate while in case of (3.14) it will. This descripancy can be resolved by producing the tokens of stream I on demand. The technique described in Section 3.7 for generating a signal stream can be used to control the production of tokens in stream I. The details are given in expression (3.16).

```
(..., S' ←(initial .
                  .
                  .
          for each i in I while p(i,x) ∧ i ≠ ε do
          .
          .
          .
          return ..., all true);
     S ← cons (true, S');
     I ← (initial i ←1
          for each s in S do
            if i ≤n then new i ← i+1;  i' ← i
                    else new i ← ε; i' ← ε
          return all i')
return ...)                                        (3.16)
```

The signal stream S must be used in generating stream I as opposed to just controlling its size after the whole stream I has been generated. For example if we write

```
I' ← (initial i ← 1
      while i ≤n do
        new i ←i+1
      return all i);
I ← exif (I', S, ε)                                    (3.17)
```

then the whole purpose of stream S is defeated.

Finally we discuss how this idea of controlled generation may be used in conjuction with a pdt. A pdt is essentially a procedure and hence can accept a stream as an input parameter and produce a stream as an output parameter. A programmer can generate an appropriate signal stream S and use it in enumerating the elements of a pdt. The technique for generating and using signal stream is same as the one described in the above examples. We think that all such programming should be explicity so that no new semantic extensions are required.

We explain by giving a very simple example where the elements of a set are to be enumerated until an element satisfies some predicate p. Let s be a pdt representing a set.

```
(! the calling procedure !
   .
   .
   .
 .., S'←(initial
          .
          .
          .
        for each x in X while ¬| p(x,Y) ∧x ≠ ε  do
          .
          .
          .
        return ..., all true);
     S ←cons (true, S' );
     X ← |enumerate| (s,S);
     .
     .
     .                    )                          (3.18)
```

Internal structure of s would look something like the following

```
procedure set (f,u,v, set-gen, a,l,S)
!all formal parameters have the usual meaning
 S - a stream parameter has been added!
(if f = ...
     .
     .
     .
 elseif f = "enumerate" then RESULT ←(initial i ←1
                                      for each signal
                                          in S do
                                       if i ≤1 then
                                          x ← a[i]
                                          else x ← ε
                                      return all x)
```

(3.19)

3.9 Mixed looping construct with a while clause

The loop of the type for each i in I, j in J while p(i,j,x) do can be compiled in two steps. First transform the sequential loop into a parallel loop according to the method discussed in section 3.2. Hence we will get

for each i in I'; j in J' while p(i,j,x) do

which can be translated either according to the rules given in [AGP78]or the rules given in section 3.7. Hence all looping

constructs involving streams can be translated.

The ideas developed so far can be suitably extended to specify efficient translation of sequential and mixed loops of the following type:

for i from 1 to n, j from 1 to m while p(i,j,x) do (3.20)

for (i from 1 to n, j from 1 to m);

(k from 1 to q, l from 1 to r) while p(i,j,x) do (3.21)

An extension of the idea present in the translation of expression (3.14) to expression (3.16) leads to the following translation of expression (3.20)

```
(...,, S' ← (initial .
                    .
                    .
          for each i in I; j in J while p(i,j,x) ∧
                                        i≠ε ∧ j≠ ε  do
                    .
                    .
                    .
          return ..., all true);
 S ← cons (true, S');
 I,J ← (initial i ← 1; j ← 1
       for each s in S do
         (if(i ≤ n) ∧ (j< m)then new j ← j+1; i' ← i; j' ← j
          elseif(i< n) ∧ (j=m) then new i←i+1 ;new j ← 1;
                                    i' ← i; j' ← j
          else i' ←ε; j' ←ε )
       return all i', all j'))                          (3.22)
```

It should be noted that in expression (3.22), the first loop satisfies property P and the moment i' and j' get the value both the loops terminate.

A translation of mixed looping construct of (3.21) is given in expression (3.23) in terms of the rules used in expressions (3.8) and (3.22).

```
(..., S' ← (initial .
                    .
                    .
             for each i in I; j in J; k in K; l in L
               while p(i,j,k,l,x) ∧ i≠ ε ∧j≠ ε∧k≠ ε ∧ l≠ ε do
                    .
                    .
                    .
             return ..., all true);
   S ← cons (true, S');
   I,J ← (initial i ← 1; j ← 1
          for each s in S do
            (if (i≤ n)∧ (j< n) then new j ← j+1; i' ← i; j' ← j
             elscif (i< n)∧(j=m) then new i ← i+1; new j ← 1;
                                      i' ← i;  j' ← j
             else i' ← ε; j' ← ε)
          return all i', all j' );
   K,L ← (initial k← 1; l← 1
          for each s in S do
            (if (k≤ q)∧(l< r) then new l ← l+1; k' ← k; l' ← l
             elscif (k< q)∧(l=r) then new k ← k+1; new l← 1;
                                      k' ← k; l' ← l
             else k' ← ε; l' ← ε)
          return all k', all l'))                          (3.23)
```

3.10 Summary

In this chapter we have modified the implementation of *for* loops from that given in [AGP78] and as explained in Chapter 2. This has been done to extend looping construct to include sequential and mixed type of loops. An explanation in Id is offered for the implementation of *for each - while* loop. This new explanation with its virtue of being in Id is applicable to many programming situations where a controlled generation of elements is required. At this stage it seems that the only fundamental looping scheme is

for each x *in* X; y *in* Y *while* p(x,y) *do*

Both the *while* loops and *for each* loops without *while* clauses can be explained as special cases of the above loop. All the other sequential and mixed looping constructs have already been defined in terms of this fundamental parallel looping construct with a while clause.

Generators as used in the literature so far can be expressed in Id without any extensions. Since pdts are basically procedures, controlled generation of elements even in case of a pdt is no problem.

CHAPTER 4

STREAM STRUCTURES

In this chapter we discuss the concept of arrays of streams, which will be called stream structures, and show their usefulness in modelling the problems related to the area of resource management. After motivating to stream structures in Section 4.1, we review the dynamic stream construct of Id in Section 4.2 and show its use by an example in Section 4.3. In Section 4.4 it is shown that stream structures can be incorporated in Id without extending the base language. We extend the semantics of some base language operators in Section 4.5 to permit streams as creation time parameters for Id monitors. Finally in Section 4.6 a complete example showing the use of stream structures is discussed.

4.1 Motivations

A problem in the area of resource management is how to maintain a queue of requests for a resource or a group of resources. In many instances a distinction among requests has to be made depending on their certain attributes, such as priority etc., to implement various scheduling policies [Bri 73, Sha 74]. In such cases many queues have to be maintained to distinguish between different requests.

Streams in Id offers a natural way to model the first-in-first-out queue*. In order to maintain several queues a stream variable can be defined for each queue. In a monitor definition in Id an entry port corresponding to each queue can also be defined. However, if the number of queues is large and dynamic addition and deletion of queues is required, then certainly, we cannot use the above idea. What we need is a model in which we can address a queue in the form A [i] , where A is the group of queues and i is the distinguishing attribute of the queue we are referring to.

The dynamic stream construct in Id [AGP77] offers a way of maintaining a variable number of queues, however, it imposes a restriction that all streams comprising a dynamic stream have to be dealt in an uniform manner. Our main motivation to go for stream structures is to provide a mechanism to model a large and a variable number of queues in which individual queue can be addressed and operations can be performed on its elements.

4.2 Dynamic streams@

As described above a dynamic stream is a set of streams whose elements are to be treated in an uniform manner. No

---

*In this chapter streams and queues will be used synonymously.

@This description of dynamic streams is taken from personal notes of Dr. Arvind and Wil Plouffe.

explicit mechanism for selecting a particular stream is provided. Id provides exactly one construct for the creation and manipulation of dynamic streams. The construct is explained in expression (4.1).

```
(dswitch  I ← S via L
 do
   J ← ( . . . )
       .
       .
       .
 dmerge  J via  M)                                    (4.1)
```

The operator dswitch creates a distinct logical stream I_j for each distinct value j in the stream L. Stream I_j will contain only those tokens of S for which the corresponding token of L has the value j. As an example

dswitch I ← [a,b,c,d,e,f,g] via [1,3,2,3,3,1,1]

would create the dynamic stream I with logical streams

$I_1 \equiv [a,f,g]$, $I_2 \equiv [c]$, $I_3 \equiv [b,d,e]$

The operator dmerge is the inverse of dswitch and creates a single stream using the dynamic stream J according to the specification stream M by removing one token from the

logical stream J_j when the next token from M carries the value j. As an example, if the value of the stream M is [2,3,3,1,3,1,1], and the stream components of dynamic stream J are

$$J_1 \equiv [a,b,c\ ],\ J_2 \equiv [d],\ J_3 \equiv [e,f,g\ ]$$

then dmerge will produce stream [d,e,f,a,g,b,c].

The operators dswitch and dmerge are deterministic in nature. The code enclosed between dswitch-dmerge pair is logically created and independently acted upon by each stream of a dynamic stream. Inside a dswitch-dmerge pair a programmer treats dynamic stream like an ordinary stream except that he cannot use ordinary streams in conjunction with dynamic streams. Below we discuss the U-interpreter semantics of the operators dswitch and dmerge.

4.2.1 dswitch:

As mentioned above dswitch creates a distinct logical stream (I_j) for each distinct value (j) in the selection stream (L). It does so by changing the context part of the activity name. We give the U-interpreter semantics of dswitch used in the dynamic stream construct of Figure 4.1.

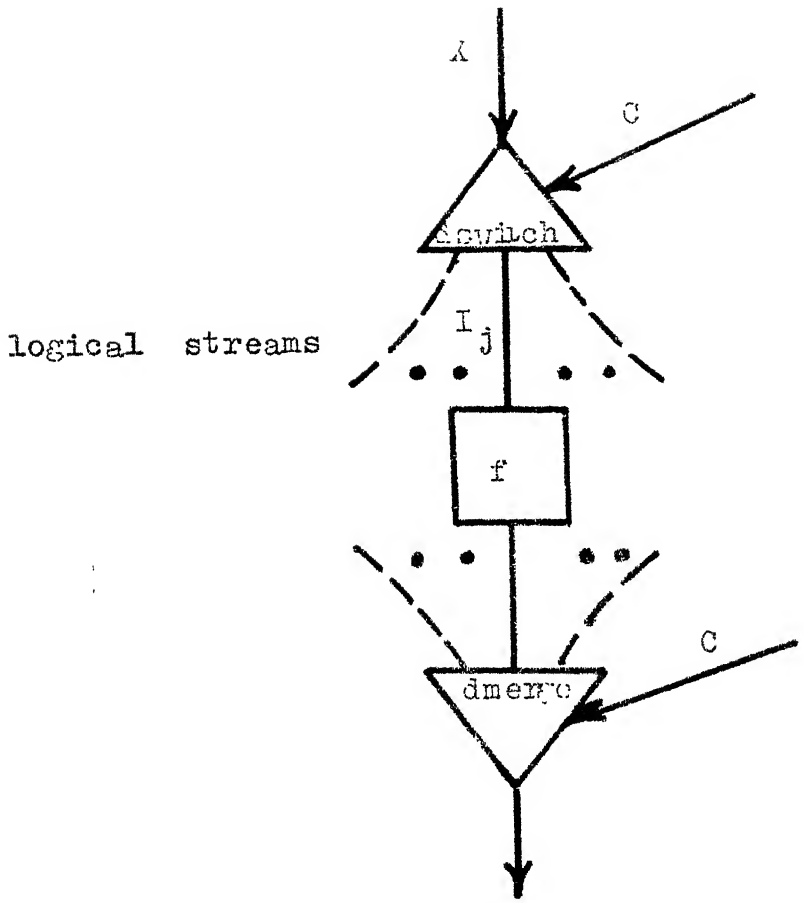


Figure 4.1 A dynamic stream construct.

u.p.s.i - - input: port1 (selection stream) $= \{\langle C_k, k\rangle \mid 1 \leq k \leq n_c\}$

port2 (data stream) $= \{\langle X_k, k\rangle \mid 1 \leq k \leq n_x\}$

output: (dynamic stream) =

$$\bigcup_{1\leq k\leq n_c} (C_k \neq \underline{est} \rightarrow (k \leq n_x \rightarrow \{\langle\langle X_k, count(C_k,k)\rangle, \langle (u.C_k).p.t.i\rangle\rangle\} ; \xi); \bigcup_{j \in DL} \{\langle\langle \underline{est}, count(j, n_c)\rangle, \langle (u.j).p.t.i\rangle\rangle\})$$

where $DL \triangleq \{j \mid j \triangleq C_k$ for some $k \leq n_c\}$

$count(j,k) \triangleq |\{i \mid C_i = j$ and $i \leq k\}|$

Thus to create logical streams context part of the activity name is changed from u to $u.C_k$. Since a logical stream is created for each distinct value of C_k, the logical streams are uniquely defined by the context part of the activity name. A logical stream is treated as an ordinary stream, and, thus it cannot have "holes" in it. For this reason dswitch maintains the number of elements passed to each logical stream by the function count. As soon as an est on selection stream is encountered, est tokens are produced for each logical stream. DL maintains all the distinct C_k values for this purpose.

4.2.2 dmerge:

As mentioned before

dmerge J via M

creates a single stream using the dynamic stream J according to specification stream M. This is done by taking an element

from the logical stream I_j when the next element from the stream M carries the value j.

We give the U-interpreter semantics of dmerge used in the dynamic stream construct of Figure 4.1. The description of dmerge is complicated by the fact that all streams of the dynamic stream have separate activity names and hence cannot be sent to the same operator. Thus dmerge has to be broken into two primitive operators: a $merge_c$ and a $merge_a$. The schematic representation of operators $merge_c$ and $merge_a$ is given in Figure 4.2.

a. $merge_c$: This operator operates on the control stream and for each token received by it, a token has to be produced by the dynamic stream construct of Figure 4.1. $merge_c$ essentially produces a stream of tokens for each of the $merge_a$ operators. The kth token received by the $merge_c$ operator carrying value c is forwarded to the $merge_a$ operator operating on the logical stream I_c to produce the kth token in the output stream.

$$u.p.s.i \;-\!-\; \text{input:(control stream)} = \{< C_k, k > \mid 1 \le k \le n_c\}$$

$$\text{output} = \bigcup_{1 \le k \le n_c} (k \neq n_c \rightarrow \{< k, \text{count}(C_k, k) >, < (u.C_k).p.merge_a.i >\};$$

$$\{< \underline{est}, k > < u.p.t.i >\})$$

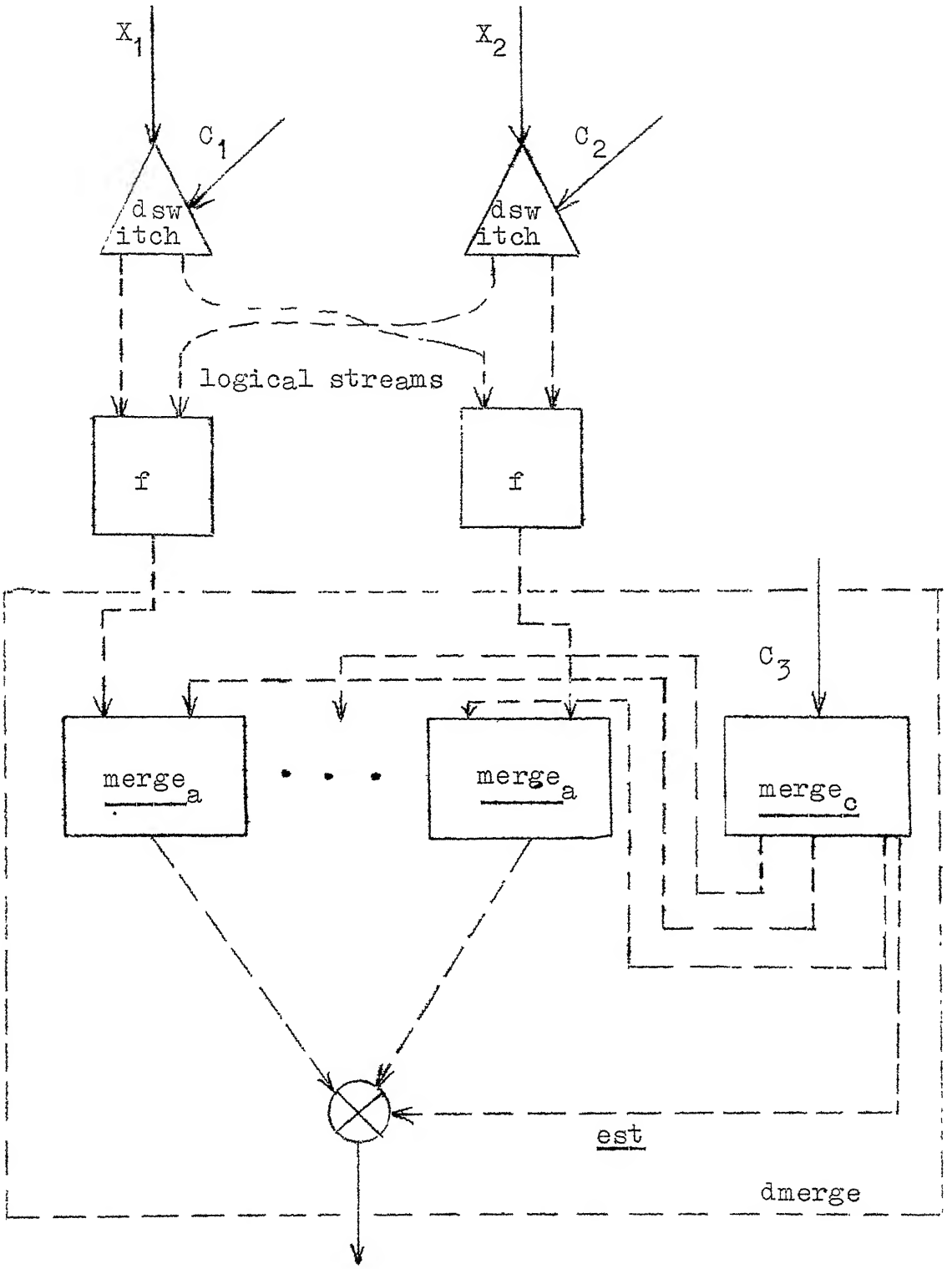


Figure 4.2 Schematic representation of merge$_a$ and merge$_c$.

Since merge$_a$ operator is identified by the context ($u.C_k$) of the logical stream on which it is operating, the context part of the activity name of the control tokens sent to merge$_a$ operators is also changed to $u.C_k$. The merge$_c$ operator has strong similarities with the dswitch operator, however, merge$_c$ produces only one final est token as opposed to an est for each merge$_a$.

b. merge$_a$: A merge$_a$ operator is needed for each logical stream I_j of the dynamic stream I. A token produced by the dmerge comes from one of the merge$_a$ operators. The merge$_a$ operator which produces the kth token of thc output stream is picked on the basis of the kth token received by the merge$_c$ operator.

$$(u.j).p.\ \text{merge}_a.i \text{ -- input: port1(control stream)} = \{<C_k, k> \mid 1 \le k \le p_c\}$$

$$\text{port2(data stream)} = \{<X_k, k> \mid 1 \le k \le p_x\}$$

$$\text{output: (stream)} = \bigcup_{1 \le k \le p_c} (k < p_x \rightarrow \{<X_k, C_k>, < u.p.t.i >\}; k = p_x \rightarrow \emptyset; \xi)$$

It should be noted that the dynamic stream construct of Figure 4.1 produces valid base language program. However,

the base language schema of Figure 4.2 will be self cleaning only under certain conditions and those conditions, in general, cannot be checked without executing the program. For self cleaning streams C_1, C_2, C_3 must have the same number of each integer present in them.

4.3 A disk scheduler monitor

We illustrate the use of dynamic streams by writing a disk scheduler. To minimize disk-head-seek-time we implement a scan policy in which disk head moves in one direction and processes all the requests that fall ahead of it. If no request lie ahead then the direction of scan is changed and the same rule is followed again. If the disk head is busy and a new request for cylinder c arrives, the counter count associated with the cylinder c is incremented. Similar to Hoare's monitor[Hoa74] we maintain a queue of requests for every cylinder with the help of a dynamic stream DREQ. The scheduler produces a stream giving the cylinder number of the next request to be processed. This stream is then used to form a dynamic stream of triggers: DTRIG, which in turn is used to send signals to queues DREQ.

Once a request is allowed to proceed, it calls on another monitor 'disk control' to actually seek the desired cylinder and transfer data. This monitor may contain some of the actual

hardware of disk controller. A reply from disk control monitor signifies that the request has been processed. The disk scheduler monitor produces a result stream RESULT which is used to signal the scheduler that the current request has been processed. A schematic of disk scheduler is given in Figure 4.3 and Id implementation is given in Figure 4.4. We will not discuss the 'disk control' monitor.

4.4 Stream structures

As pointed out earlier, dynamic stream construct provides no way of selecting a particular logical stream. Consider a slight modification of the problem of disk scheduling discussed in Section 4.3. Suppose the disk scheduler should access two different physical disks depending on which group does the cylinder number belongs to. Solution of the problem using the dynamic stream would require the use of a case statement operating on all logical streams thus no longer maintaining problem structure in the solution. This solution is also likely to be inefficient in some cases. We will discuss these points further in Section 4.6.

In fact the dynamic stream construct in Id solves two problems associated with the resource management.

1. The dswitch operator is used to form large and variable number of queues. In doing so initial ordering of requests is maintained only within a particular queue.

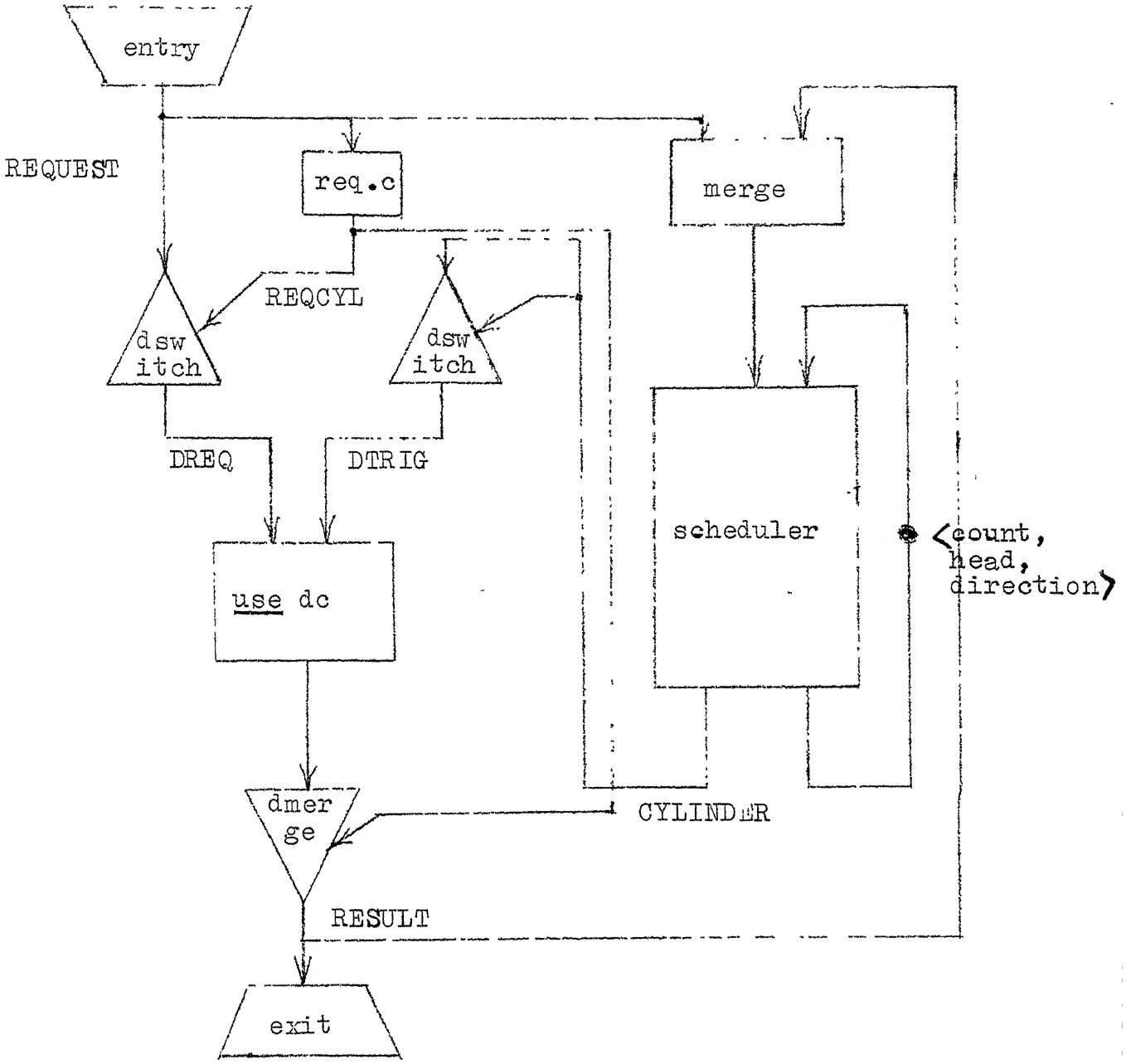


Figure 4.3 Schematic of disk scheduler monitor of Figure (4.4).

```
monitor disk scheduler (dc)
! dc is the disk control monitor for the disk this scheduler
  is supposed to schedule!
(entry REQUEST do
       REQCYL ← (for each req. in REQUEST do
                 return all req.c);
       ! req.c gives the cylinder number!
       RESULT ← (dswitch DREQ ←REQUEST via REQCYL;
                         DTRIG ← CYLINDER via CYLINDER
                 do
                         DRESULT ←(for each dreq in DREQ;
                                          t in DTRIG do
                                    dres ← use(dc,dreq)when t
                                  return all dres)
                 dmerge DRESULT via REQCYL);
       X ←merge (REQUEST, RESULT);
       ! This is scheduler code!
       CYLINDER ← (initial busy ← false; direction ←"up";
                           head ← ∅
                   for each x in X do
                     if x.type = "request"
                     then (if busy
                           then new count [x.c]← count [x.c ]+1;
                                cylinder  ← λ
                           else new direction ←(if x.c > head
                                                then "up"
                                                else "down" );
                                new head  ←c;
                                new busy  ←true;
                                cylinder  ←c)
                     else ! it is a done signal!
                      new count, new head, new direction,cylinder
                            ←findnext (count,head,cylinder);
                      new busy ←(if cylinder= λ then false
                                                else true))
                   return all cylinder but  λ)
  exit RESULT)
```

Figure 4.4 Code for disk scheduler monitor.

2. The dmerge operator is used to restore the initial ordering by choosing the appropriate control stream (see the use of stream REQCYL in Figure 4.4).

Another solution to maintain a variable number of queues in which individual queue can be addressed is to introduce array of streams - which will be called as Stream Structure.

A stream structure is a stream in which each element has an associated qualifier which will be called "selector". The selector can be part of the activity name or part of the data itself. For further discussion we will assume it to be the part of data. Thus no new Id constructs or base language operators are required to deal with stream structures. An element of the stream structure is of the form:

<<< "data": x, "selector": int >, k > , < activity name >>

where int is a selector value. We will assume all selector values to be integers. A stream structure can also be regarded as an ordinary stream of structure values. Hence, all operators and constructs defined on simple streams are valid on stream structures also. In addition we define two Id procedures which are useful when dealing with stream structures.

a. select (A,i): A is a stream structure and i is a selector value. The result of procedure invocation is a stream containing those elements of A whose associated selector value equals i. Expression (4.2) gives details of the procedure.

```
procedure select (A,i)
(for each a in A do
   res ← (if a [ "selector" ] = i then a [ "data" ]
                                  else λ)
 return all res but λ)                                    (4.2)
```

b. append (A,B,i): A is a stream structure, B is a simple stream, and i is a selector value. Result of procedure invocation is a stream structure containing those elements of A whose selector value is not equal to i and all elements of B with their selector value made equal to i. Expression (4.3) gives details of the procedure.

```
procedure append (A,B,i)
(RES1 ← (for each a in A do
            res1 ← (if a [ "selector" ]= i then λ
                                           else a)
         return all res1 but λ);
 RES2 ← (for each b in B do
            res2 ← < "data": b, "selector":i >
         return all res2)
 return merge (RES1, RES2))                               (4.3)
```

merge in the return clause of the procedure append is used to cater for infinite streams. It can be replaced by concatenate operator, if only finite streams are being considered

The use of merge makes the procedure append nondeterministic in behaviour. However, if we allow only for each loop and the procedure select to operate on stream structures, then either all the elements can be dealt in an uniform manner or a particular stream can be selected. Since merge does not change ordering of elements in component streams, the selected stream is always deterministic. Thus under these limitations the ultimate result will always be deterministic.

4.5 Extension of Id monitors to include streams as creation time parameters

The problem of maintaining queues is solved using stream structures. However once selection of a particular stream of a stream structure is done, the position of the elements of the selected stream within the original stream is lost. However, for correct functioning of Id monitors, it is required that a strict correspondence between the elements of input and output streams be maintained*. Hence, for stream structures to be useful in resource management problems,we need a way to restore the original order. We extend the monitors to include streams as creation time parameters and then we will

---

*Note if this is not the case, that is, the answer corresponding to the ith token of the input stream is in the jth position ($j \neq i$) of the output stream then the sender of the ith token will get the wrong answer (i.e., the ith token from the output stream).

use the extended monitor and dynamic streams to alter the ordering of elements in a stream structure.

We will allow the streams to be parameters in the construct create (see Section 5.2 of [AGP78]). For each stream parameter in create we provide separate port in operators C_2 and MBEGIN. A schematic of such extension is shown in Figure 4.5. Below we give the activity name manipulation by the operators C_1, C_2 and MBEGIN.

i. C_1: This operator needs no change to accomodate streams as creation time parameters.

$u.c.s_1.i$ - - input = c_m

output: port1= $<(u'.c_m.mbegin.1),$
$<u.c.s_2.i,1>>$

port2= $(u'.c_m.entry.1)$

ii. C_2: A separate port is created for each stream parameter.

$u.c.s_2.i$ - - input: port1= $(u'.c_m.mbegin.1)$

port2= x

port3(stream)= $\{<y,k> \mid 1 \leq k \leq n_y\}$

output: port1= $<x, <u'.c_m.mbegin.1,1>>$

port2(stream)=$\{<<y,k>$
$<u'.c_m.mbegin.1,2>>$
$\mid 1 \leq k \leq n_y\}$

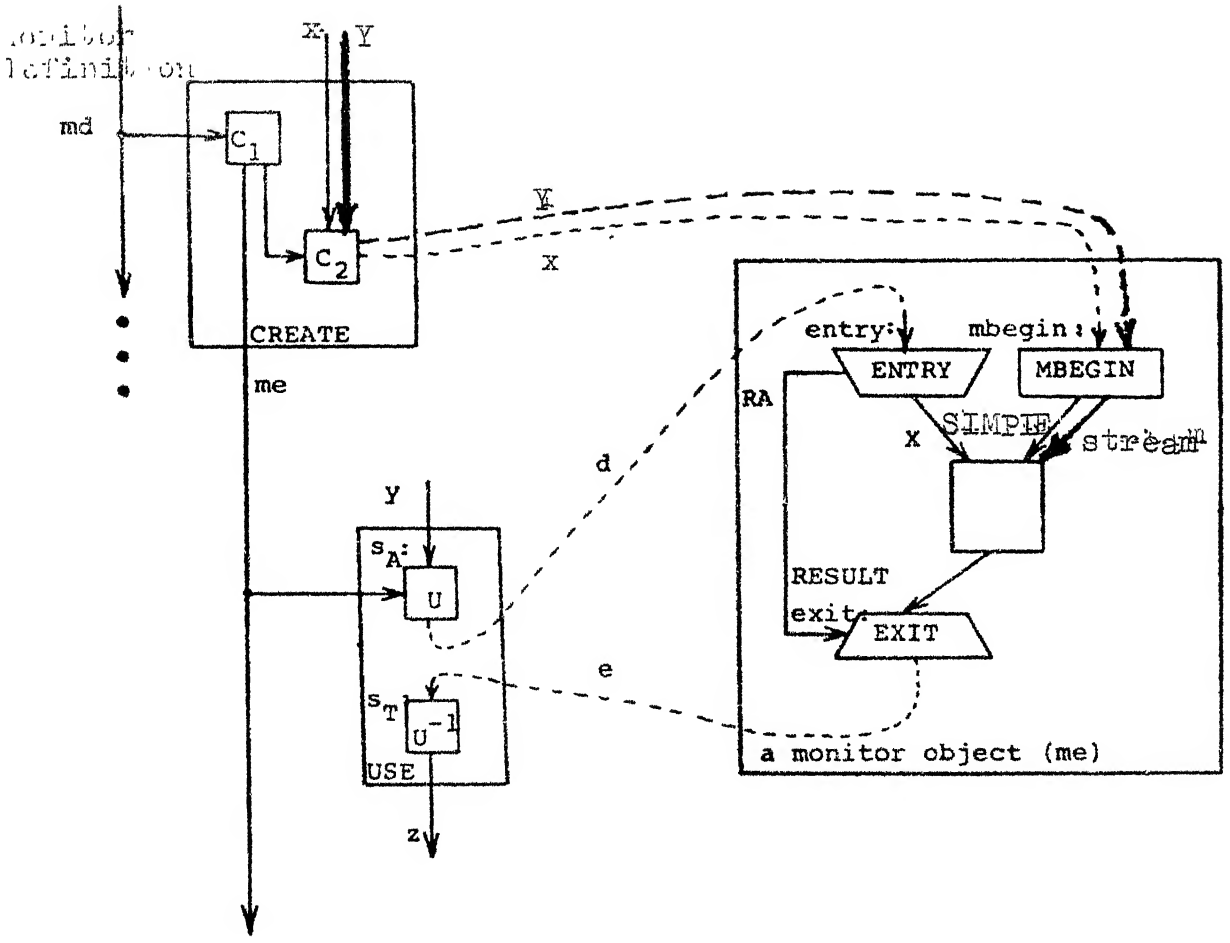


Figure 4.5 Schematic of extended monitor.

Thus the operator C_2 will remain in existence until it either receives eot tokens on all streams or the monitor object is destroyed.

iii. MBEGIN:

$u'.c_m$.mbegin.1 -- input: port1(simple)=x

port2(stream)=$\{< y,k> | 1 \leq k \leq n_y\}$

output: port1(simple)=x

port2(stream)= $\{< y,k > | 1 \leq k \leq n_y\}$

A monitor to alter the ordering of elements of a stream structure is given in expression (4.4).

```
monitor nexts (STREAM-ST)
(entry SELECTOR do
      RESULT ← ( dswitch DSEL ← SELECTOR via SELECTOR;
                        DSTREAM-ST ← STREAM-ST via
                                 (for each s in
                                       STREAM-ST do
                                 return all s["selector" ])
                 do
                        RES ← (for each ds in DSTREAM-ST;
                                       dsel in DSEL do
                                 return ds when dsel)
                 dmerge RES via SELECTOR)
 exit RESULT )                                          (4.4)
```

Use of an instantiation of such a monitor by use (dnexts,c), where dnexts is a monitor object created from monitor definition nexts, will give the next element in the logical stream DSTREAM-ST_c. This is in fact, the next element whose selector value is c in the stream supplied to dnexts at creation time.

The effect of monitor nexts can also be achieved by providing an entry port for stream STREAM-ST without extending the monitors to include streams as creation time parameters. Such a version of monitor nexts is given in expression (4.5)

```
monitor nexts ( )
(entry selector: SELECTOR; data: STREAM-ST do
      RESULT1 ← (dswitch DSEL← SELECTOR via SELECTOR;
                         DSTREAM-ST ←STREAM-ST via
                                  (for each s in
                                        STREAM-ST do
                                   return all s[ "selector"])
                 do
                         RES← (for each ds in DSTREAM-ST
                                    dsel in DSEL do
                               return all ds when dsel)
                 dmerge RES via SELECTOR);
       ! we also need to acknowledge the processes sending
         elements of STREAM-ST at entry port "data"!
       RESULT2 ← (for each s in STREAM-ST do
                  return all 'acknowledged')
 exit selector: RESULT1; data: RESULT2)                    (4.5)
```

The stream STREAM-ST is sent to an instance of nexts element by element as explained in program segment of expression (4.6).

```
dnexts ← create (nexts, )
(initial flag ← anything
 for each s in STREAM-ST do
   ack ← use (dnexts.data,s) when flag;
   new flag ←ack
 return ... )                                    (4.6)
```

Due to unfolding of loops in Id second iteration of loop in expression (4.6) may produce s before first iteration does so. Since the entry operator is nondeterministic, it will alter the ordering of elements in STREAM-ST. To avoid this we used a flag in expression (4.6). Flag will get a new value for the next iteration only when the previous iteration has completed the use of the monitor, thus forcing serial use of monitor instance dnexts. We want to emphasize the fact that any time a monitor is to be used sequentially it should be explicitly programmed by the programmer. Actually we are tempted to introduce a new U operator in the base language that will automatically ensure sequentiality in the use of a monitor. This new operator will receive an acknowledge signal from the entry of the monitor as soon as a token is

received. The advantage is, of course, one does not have to program explicit acknowledge signals, generating which can be quite clumsy in some cases.

Though monitor nexts in expression (4.5) is equivalent to monitor nexts in expression (4.4) as far as overall behaviour is concerned, monitor in expression (4.4) reflects the nature of the problem more clearly than the monitor in expression (4.5). Also limiting creation time parameters of a monitor to be simple variables is really not necessary.

Inside monitor nexts we use the dynamic streams to maintain queues and alter ordering. Such a use of dynamic streams offers two advantages:

i. Dynamic streams can be made accessible only to monitor nexts thus avoiding the problem of unintentional mixing of dynamic streams with simple streams by users. This also gives a kind of desirable modularity to the program.

ii. Semantics of monitor nexts is easier to understand than the semantics of a general dynamic stream construct. This is due to the fact that net effect of monitor nexts is only to alter the ordering unlike dynamic streams which perform two functions at a time.

As a passing point we would like to mention that the concept of monitor nexts can be applied to model next construct on streams. Expression (4.7) gives a monitor whose behaviour is equivalent to construct next.

```
monitor  next (X)
(entry REQ do
        Y ← exif (X,REQ, ε );
        RES ← (for each req in REQ; y in Y do
                return all y)
 exit RES )                                        (4.7)
```

As an example consider an Id program which uses the next operator.

```
Z ← (for each b in B do
       z ←(if b then next X
                else next Y)
     return all z)                                 (4.8)
```

The above program can be written using monitor next as:

```
xnext ← create (next,X);
ynext ← create (next,Y);
Z ← (initial xsig← anything; ysig ←anything
    for each b in B do
      if b then z ←use (xnext, xsig);
                new xsig ← z;new ysig ← anything
           else z ← use (ynext, ysig);
                new xsig ← anything; new ysig ← z
    return all  z)                                 (4.9)
```

4.6 An example

Consider a multiprogrammed, multiprocessor environment. Suppose each processor is capable of performing several different tasks, and the set of tasks performed by any processor is exclusive of the sets of tasks performed by all the other processors. There are several requests for each type of task. We want to write an interface monitor which will schedule the tasks on the processors. We will assume that there are only two processors and different task types are numbered from 1 to 2n with 1 to n running on processor 1 and n to 2n running on processor 2. Real world examples of such a system would be a system like CRAY 1 [Rus78] having separate processors for scalar and vector operations or a system in which one processor is dedicated to string operations (compilation etc.) and one processor to arithmetic calculations (floating point arithmetic unit). Each task type has a priority associated with it, and the number of requests waiting in its queue. These numbers are maintained in two arrays: priority and count. A procedure "change-priority" changes the priority of tasks dynamically depending on the number of requests waiting. Another procedure "highest-priority" gives the highest priority task number, whose queue is not empty. The number of the highest priority task is between 1 to n or n to 2n

depending on whether the flag is set to 1 or 2. If no such task is found then the procedure "highest-priority" returns λ. Each processor has an associated monitor processor[i] ; i=1,2 which may be the actual hardware of the processor. This monitor returns its number i in result. "processor" to interface monitor when it completes the scheduled task. A complete description is given in expression (4.10). We maintain separate queues for each type of task using stream structure ST-REQ.

```
monitor interface (processor, count, priority)
(entry REQUEST do
        ! streams REQUEST and TRIG are changed to
          stream structures!
        ST-REQ ←(for each req in REQUEST do
                    streq  ← <"data":req,
                              "selector": req. task>
                 return all streq);
        ST-TRIG ← (! TRIG is returned by scheduler code !
                    for each trig in TRIG do
                      sttrig ← <"data": trig,
                                "selector": trig >
                    return all sttrig);
```

```
RES1 ← (for i from 1 to n do
          ST-REQ1 ← select (ST-REQ, i);
          ST-TRIG1 ← select (ST-TRIG, i);
          RRES1 ← (for each streq1 in ST-REQ1;
                                sttrig1 in ST-TRIG1 do
                     result ←use (processor[ 1],
                                  streq1)
                                when sttrig1;
                     rresult ← <"data": result,
                                "selector":i >
                   return all rresult)
        return all RRES1);
RES2 ← (for i from n to 2n do
          ST-REQ2 ← select (ST-REQ, i);
          ST-TRIG2← select (ST-TRIG, i);
          RRES2 ← (for each streq2 in ST-REQ2
                                sttrig2 in ST-TRIG2 do
                     result ← use (processor [2],
                                  streq2)
                                when sttrig2;
                     rresult ←< "data": result,
                                "selector": i>
                   return all rresult)
        return all RRES2);
RESULT ← merge (RES1, RES2);
X ← merge (REQUEST, RESULT);
```

```
! following is scheduler code!
TRIG ← (initial...
        for each x in X do
          if x.type = "req"
          then (if x.task≤ n
                then(if status [processor [1]]="free"
                     then trig ←.x.task
                     else new count [x.task] ←
                              count [x.task] +1;
                          change-priority (priority,
                                           count);
                          trig ← λ)
                else (if status [processor [2]]="free"
                      then trig ← x.task
                      else new count [x.task] ←
                               count [x.task] +1;
                           change-priority (priority,
                                            count);
                           trig ← λ))
               else ! it is a done signal from processor
                      monitor!
                    (if x."processor"=1
                     then trig ← highest-priority(
                                  priority, count,1);
                          status [processor [1]] ←
                           (if trig= λ then "free"
                                       else "busy")
                     else trig ← highest-priority (
                                  priority, count,2);
                          status [processor [2]] ←
                           (if trig= λ then "free"
                                       else "busy"))

         return all trig but λ) ;
```

```
        ! following is code for restoring ordering!
        rnexts ← create (nexts, RESULT);
        EX-RES ← (initial flag ← anything
                  for each req in REQUEST do
                    exres ← use (rnexts, req) when flag;
                    new flag ← exres
                  return all exres)
    exit EX-RES)                                          (4.10)
```

Procedures "change-priority" and "highest-priority" are easy to write and we will not discuss them here.

The monitor interface,if written using dynamic streams will look something like:

```
monitor interface (processor, count, priority)
(entry REQUEST do
      REQ ← (for each req in REQUEST do
             return all req. task);
      RESULT ← (dswitch DREQ ← REQUEST via REQ;
                        DTRIG ← TRIG via TRIG
                do
                  RES ← (for each dreq in DREQ;
                                  dtrig in DTRIG do
                            res ← (if dreq. task ≤ n
                                      then use(processor[1],
                                               dreq)...
                                   elseif dreq.task > n
                                          dreq.task ≤ 2n
                                          then use
                                          (processor[ 2],
                                           dreq)...
                                   else 'error' )
                                .
                                .
                                .          )
                dmerge RES via REQ)
    exit RESULT)                                          (4.11)
```

In expression (4.10) the stream structures ST-REQ and ST-TRIG circulate in the loop. However, since input and output of streams is asynchronous, and ST-REQ and ST-TRIG are not altered by the loop, all the instantiations of inner loop will proceed simultaneously. Thus there is no substantial loss of asynchrony (only that the last instantiation of the inner loop will start after n units of time). The program in expression (4.10) uses the select operation on stream structures whose execution time may be comparable to the execution time of case statement used in expression (4.11), when the number of conditions are less (say 2 or 3). However, with a large number of conditions execution time of the case statement may be too high, and, it will also be cumbersome to write the predicates. The program in expression (4.10) does reflect the steps in the computation more clearly than the program in expression (4.11).

4.7 Summary

We have discussed stream structures which are useful in maintaining large and variable number of queues in the resource management. In fact stream structures can be used in any place where arrays of streams are required. Furthermore stream structures require no new Id constructs and/or base language operators and they are completely user-programmable.

CHAPTER 5

PROGRAMMER DEFINED CONSTRUCTS

In this chapter we propose a mechanism in Id by which new constructs can be defined. We call it programmer defined construct. Such a mechanism is useful in abstraction of operations and in defining one's own language. In Section 4.2 we discuss the syntax for the definition and the use of a programmer defined construct and the implementation details. We clarify certain points by discussing several examples in Section 4.3. Finally, in Section 4.4 we discuss the limitations of our approach.

5.1 A case for programmer defined constructs

Design and implementation of large and complex software systems, their subsequent maintenance and modification, and proving correctness of such systems are facilitated if we reduce the amount of complexity that must be considered at any one time. Abstraction, whether it is of operations (events) or data, provides a way to separate "what" from irrelevent "how" and thereby reduce the complexity. Data abstraction [Gut77,, LSAS77, SWL77, WLS76] has drawn much attention recently. However, we would also like to view operations in an abstract way. To some extent, procedures in programming languages provide a way to the abstraction of operations. A procedure can be treated as a black box which performs a function and at

the time of its use one need not worry about "how" it does so.

However, procedures are static in their nature, in the sense that they can operate only as a particular expression over different sets of parameter values. A more dynamic facility would be one which specifies only the relationship among the expressions, the actual expressions being supplied at the time of its use [Bac78].

As we mentioned in Chapter 1 that most of the existing programming languages are becoming bulky with the addition of newer and newer constructs to deal with specialized situations or extensions. Still, they remain inherently weak because they can not deal with situations for which they are not designed. This is due to the defects at the most basic level in their design. Their semantics is closely coupled to state transition and they divide the programs into expressions and statements. Whereas the expressions can be combined very easily to form higher level programs, it is difficult to do so with the statements. The result is that conventional languages are forced to have more rigid parts and less variable parts. A more rational approach would be to provide primitive building blocks alongwith a facility which is helpful in building complex and specialized structures using these primitive building blocks. Thus one should be

able to create his own language tailored to his needs [Bac78].

Bearing the above two points in mind we propose a mechanism in Id which allows one to define his own constructs. We call them programmer defined constructs (pdcs in short). Such mechanisms are provided in LISP [McC62] and RED languages [Bac73, Bac78]. CLU [LSAS77] and ALPHARD [SWL77] provide an abstract view of looping mechanisms. A dataflow language CAJOLE [HOS78], though in a very preliminary development stage, incorporates the idea of programmer defined constructs.

5.2 Programmer defined constructs in Id

A programmer defined construct in Id is composed of two parts - its definition and its use. The definition of a pdc specifies the relationship among the expressions (to be supplied later at the time of its use) using valid Id schemas. In defining the relationship, place of an expression is denoted by a variable enclosed in { }. We will call these variables exp - variables. For present discussion we assume that only three Id schemas - block expressions, conditional expressions, and loop expressions, can be used inside the definition of a pdc. The use of a pdc specifies the actual expressions and the values of the arguments. These expressions are substituted in place of exp variables in the definition of a pdc. This

substitution is performed at run time so the recursive definitions are allowed. The expression obtained after substitution is converted to a procedure value which is then applied to the actual argument values specified in the use of the pdc. This technique of defining constructs has some similarities with the well known conditional assembly of macros used in low level languages [Str65].

5.2.1 Definition of a pdc:

The syntax for defining a pdc is given informally in expression (5.1).

construct [{cplist}] ⟦ "≪construct word≫" {≪body≫} ⟧
=(≪definition of the construct≫)

(5.1)

⟦ , ⟧ , ≪ , ≫ are metasymbols. ⟦ ⟧^{n} denotes n occurances of the string enclosed in ⟦ ⟧ and for our purpose $n \geq 1$. The formal syntax of construct definition is given in Figure 5.1.

The string ⟦ "≪construct word≫"{≪body≫}⟧^{n} specifies how the pdc will be written when it is used in a program. The exp-variable { cplist } denotes a list of variables which will be supplied by an use of the pdc. We want to emphasize here that only valid Id expressions, and not Id statements, can be substituted in place of exp-variables i.e. variables enclosed

≪construct definition≫ ::= construct [≪exp-variable≫]
≪usage form≫ (≪construct def≫)

≪exp-variable≫ ::={≪identifier≫}

≪usage form≫ ::=≪construct word-body≫≪usage form≫ [] ≪construct word-body≫

≪construct word-body≫ ::=≪construct word≫≪exp-variable≫

≪construct word≫ ::= "≪identifier≫"

≪construct def≫ ::=≪valid extended bclu Id expression≫

≪valid extended bclu Id expression≫ ::= valid Id block, conditional or loop expression or a construct use with ≪extended Id variables≫

≪extended Id variables≫ ::= ≪Id variable≫ [] ≪exp-variable≫

≪Id variable≫ ::= ≪identifier≫

≪identifier≫ ::= string of English alphabets and special symbols excluding null string.

Figure 5.1 Syntax of the definition of a pdc.

in { }. As an example, we give definition of a case expression using conditional expression in expression (5.2).

```
construct [ {cplist} ] "case |"{pred 1 }" ⇒ "{ body 1 }
                              ""|"{pred 2 }" =>"{ body 2 }

≡(if {pred 1} then {body 1 }
  elseif { pred 2 } then {body 2}
  else      )                                    (5.2)
```

Expression (5.2) does not bring out all the points which are important, however, it is only intended to give a flavour of syntax. Other points will be made clearer in later sections.

5.2.2 Use of a pdc:

The syntax for the use of a pdc can be explained informally by expression (5.2).

```
[ <<list >> ][" <<construct word >>" <<exp >>]]^n
using [<< argument list >> ]                      (5.3)
```

Formal synatx is given in Figure 5.2. 'exp' denotes any valid Id expression or a list of expressions. 'list' specifies the variables used in these expressions. Actual values of these variables should be supplied from outside. We have introduced a new reserved word, using. The list following using i.e. 'argument list' specifies the actual values for

≪ construct use ≫ ::= [≪ list ≫] ≪ use form ≫

using [≪ argument list ≫]

≪ list ≫ ::= ≪ identifier ≫, ≪ list ≫ ‖ ≪ identifier ≫

≪use form ≫ ::= ≪ construct word use ≫ ≪ use form ≫ ‖

≪ construct word use ≫

≪construct word use ≫ ::= ≪ construct word ≫ ≪valid Id exp+const ≫

≪valid Id expression+const ≫ ::= ≪ valid Id expression ≫ ‖

≪ construct use ≫

≪argument list ≫ ::= ≪ valid Id expression ≫ , ≪ argument list ≫

‖ ≪ valid Id expression ≫

≪construct word ≫ ::= "≪ identifier ≫ "

≪valid Id expression ≫ ::= any valid Id expression

≪identifier ≫ :.= string of English alphabets and special symbols excluding null string

Figure 5.2 Syntax of the use of a pdc.

variables listed in 'list'. The correspondence is by position (easiest, though other formulations can be specified). 'Argument list' may contain valid Id expressions as the values for the variables listed in 'list', in which case expressions are first evaluated and then the evaluated values are passed. using also specifies the scope of the pdc.

Expression (5.4) shows the use of the pdc defined in expression (5.2).

```
z ← ( [x,y,a,b,i,n] "case |" i< n"=>" x+2*a
                    "|" i= n"=>" y+2*b*a
    using [x,y,e,f,i,10] )                    (5.4)
```

Here we would like to point out the difference between x in 'list' and x in 'argument list'. x in 'list' denotes a variable which will be used inside the pdc, whereas, x in 'argument list' denotes the variable (i.e., the line in base language graph) outside the pdc. using specifies an interconnection between them. The facility of using can be made general enough so that it can be used to avoid writing statements in many cases.

5.2.3 Implementation of a pdc:

Like a procedure value, the definition of a pdc is translated as constant function which produces a new type of

value called 'pdc'. The value of the type pdc has the following internal representation:

< name: pdc name,
usage form: the string enclosed in ⟦ ⟧ in expression (5.1)
definition: definition of the pdc >

pdc name is obtained by concatenating all the construct words. The translation of expression (5.2) is shown in Figure 5.3.

The use of a pdc is translated as a function evalc. The inputs to the function evalc are:

1. pdc definition value.
2. A structure containing 'list' and the actual expressions in the form of strings. For future reference this structure will be called expvalue.
3 The actual values of arguments as given in 'argument list'.

The function evalc evaluates the pdc and its outputs are the outputs of the pdc expression. Figure 5.4 shows the translation of expression (5.4).

The function evalc is not implemented as a primitive operator. It is composed of two operators - substitute, and apply. Figure 5.5 shows the internal structure of evalc and one may refer to it to understand the functions of two operators, which are discussed below.

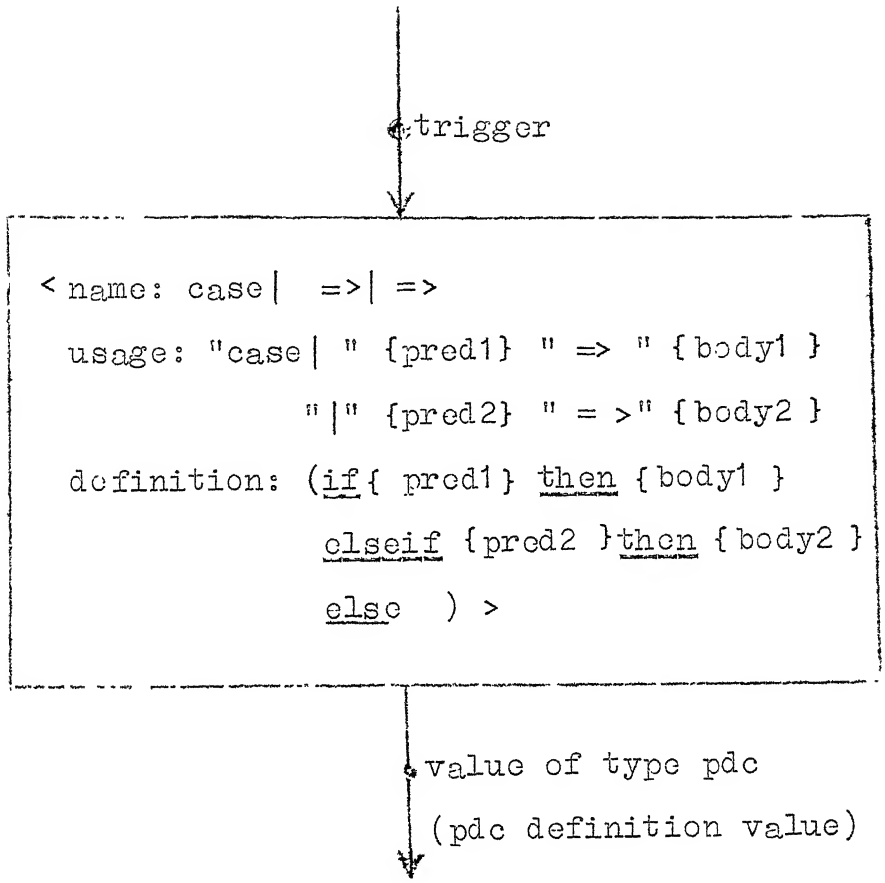


Figure 5.3 Translation of the definition of pdc of expression (5.2).

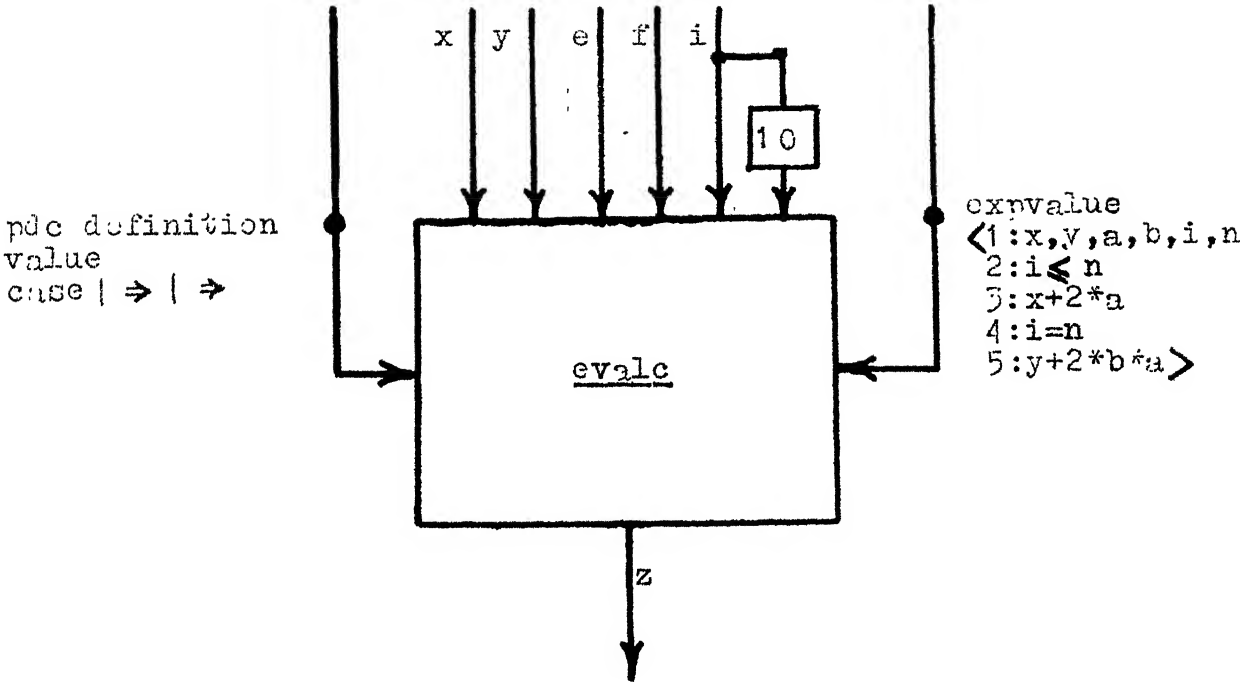


Figure 5.4 Translation of the use of a pdc (expression (5.4)).

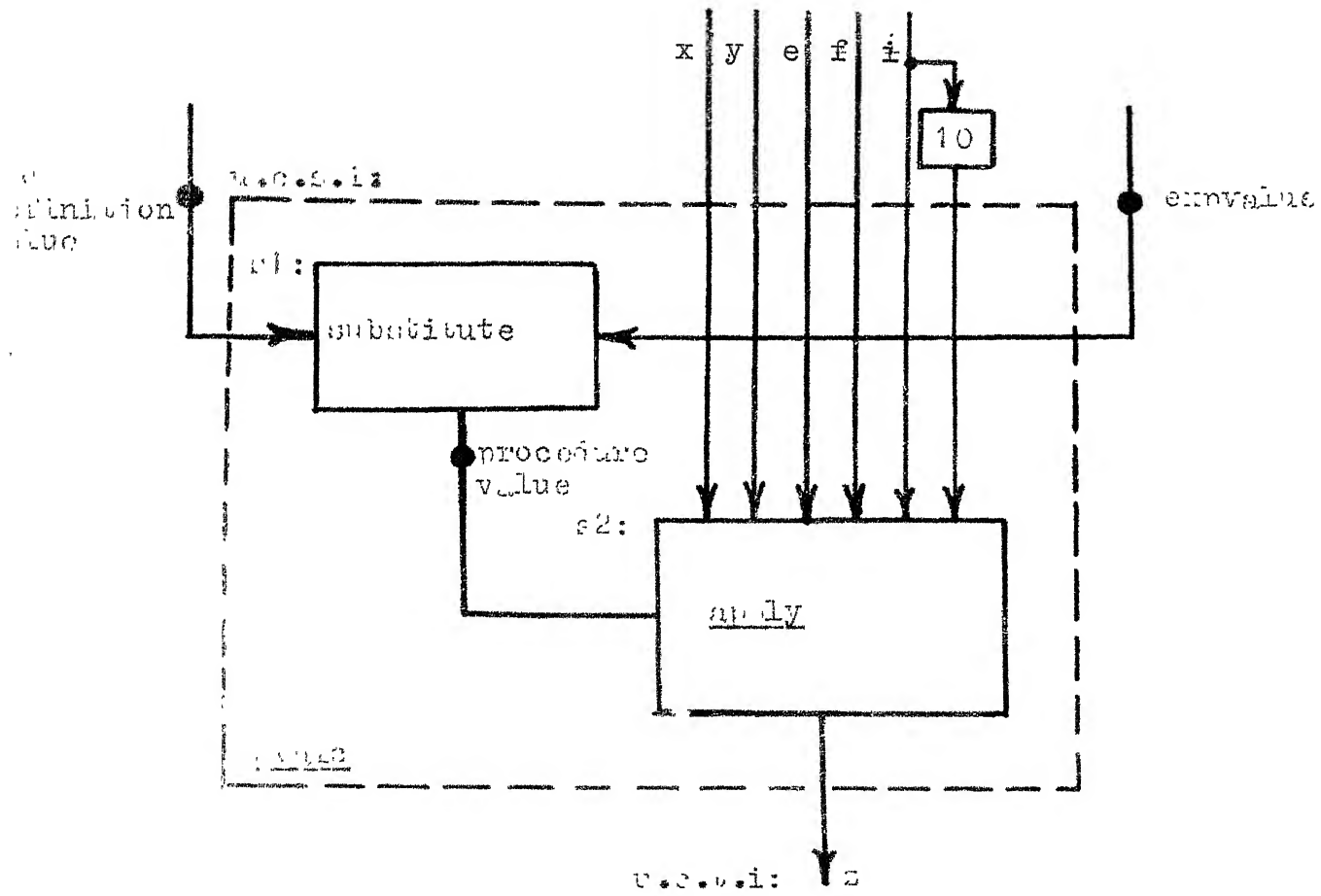


Figure 5.5 Schematic of <u>cycle</u>.

a. substitute:

It performs two functions - it substitutes the actual expressions in place of exp-variables and then it creates a new procedure value using substituted definition. If the data structure of Id expressions is represented in the form of structure or string values then no new operator except a coercion function from structure or string to procedure is required. The created procedure value is passed to the apply operator. The actual expressions are obtained from structure expvalue. The correspondence between actual expressions and exp-variables is decided using usage part of pdc definition value and is by position i.e. expvalue.1 corresponds to {cplist} and so on. Name of the created procedure value is the first construct word and formal parameter list of the procedure value is the actual list corresponding to {cplist}. To allow for recursive definition of a pdc, pdc definition value is passed to the procedure value as a "freezed" argument.

$u.c.s_1.i$ -- input: port 1 (pdc definition value) = c_v

port 2 = expvalue

output = << create proc (subst(c_v, expvalue)) >,

< $u.c.s_2.i$ >>

subst performs string substitution and it can be expressed as an Id procedure. However, createproc converts

the substituted definition to a procedure value and cannot be expressed in Id. The procedure value passed to apply operator at the execution time of expression (5.4) is given by q in expression (5.5).

```
p ← procedure case (construct, x,y,a,b,i,n)
    (if i < n then x+2*a
     elseif i = n then y+2*b*a
     else      );

q ← compose (p,< case | =>|=> ,,,,,, > )        (5.5)
```

b. apply:

It takes the procedure value which is passed to it by the operator substitute and applies it to the actual arguments. The results are passed to the program segment where construct is used. This operator is discussed in some detail in Section 2.3.

5.3 Some examples

In this section we will discuss some examples of pdcs to illustrate some of the points relevant to their definition and their use.

a. A while loop:

Expression (5.6) gives recursive definition of a pdc which is equivalent to the while loop in Id. Expression (5.7) shows its use.

```
construct [{ cplist} ] "while"{ predicate} "do"
                                    { body }
        ≡ (if { predicate } then [{ cplist }]
                                        "while"{predicate }"do"
                                              {body}
                                        using [{body}]
                                  else {cplist} )        (5.6)

[sum, x,i,n ]"while" i≤n  "do"
                          sum +x,x,i+1 ,n
using [sumo, xo,1,10 ]                                   (5.7)
```

Values returned by expression (5.7) are the final values of the expressions sum+x, x, i+1, n. This example brings out a restriction i.e. use of any recursively defined pdc should return the values for all the variables specified in its 'list'.

b. A for-while loop:

We implement a for-while loop using the pdc mechanism and the for each-while loop. The implementation is based on the translation of for-while loop into a for each-while loop as discussed in Chapter 3. The definition of the construct is given in expression (5.8).

```
construct [ {cplist} ]"for"{var} "from" { lower limit }
                  "to" {upper limit }"by" { step }
                  "while" { pred. }      'do"
                          { body }
                  "return" { explist }
        ≡ ('local'.{explist}, S' ←(initial{cplist }←{cplist }
                                    for each{var}in I
                                    while {pred}∧{var}≠ ε do
                                      new({ cplist}) ←{ body}
                                    return{explist},all true);
          S← cons (truc, S' );
          I← (initial j ←{lower limit }
              for each s in S do
                if j≤{ upper limit}then new j ← j+{step};
                                        j' ← j
                                   else new j ←ε ;
                                        j' ← ε
              return all j' );
          return 'local'. {explist } )                    (5.8)
```

The use of the above construct is shown in expression (5.9).

```
[ sum, p,q,r,n,a ] "for" i "from" p "to" q "by" r
                   "while" sum < n "do"
                           sum+i*a,p,q,r,n,a
                   "return" 3*sum
using[ sumo, 1,10,2,s,b ]                          (5.9)
```

We used the reserved word new in a rather different manner in expression (5.8). If we assume that { cplist } ≡ a,b,c then at the time of substitution new ({ cplist }) will be converted to new a, new b, new c. We have also introduced a new way of generating variable names. 'local'.{ explist } means that string 'local' is concatenated with all the strings occuring in {explist} and new variable names are formed. As an example, if {explist} ≡ x,x+y then

'local'. { explist } ≡ localx,localx+y.

localx+y is the name of a variable. The need to do this stems from the single assignment property of Id.

c. A for loop that returns streams:

Expression (5.10) gives the definition of a pdc which is equivalent to the Id for loop with return all construct. Expression (5.11) shows how it can be used in a program.

```
construct [{cplist }] "for"{var} "from" {lower`limit}
                                    "to"{ upper limit}"do"
                              { body }
                          "return"{ explist 1 }"all" {explist 2}
        = (initial{ cplist }←{cplist}; { var }←{lower limit }
           while {var }≤ {upper limit } do
             new( { cplist } ) ← { body };
             new  {var}  ←{ var } + 1
           return {explist 1 } , all ( {explist 2}))
                                                        (5.10)

[x,y,p,n] "for" i "from" p "to" n "do"
                f(x,y,i), f'(x,y,i), p,n
          "return" x,y "all" g(x,y,i),g'(x,y,i)
using [xo,yo,1,10 ]                                    (5.11)
```

In this example we used a *while* loop to define a *for* loop. In expression (5.10) the use of *all* is similar to the use of *new* in expression (5.8).

d. *A sequential loop construct*:

We now implement a sequential *for each* loop construct as a pdc. Our implementation is based on translation given in Chapter 3 (expression (3.5)). The definition of the pdc is given in expression (5.12) and its use in expression (5.13). This example brings out some of the limitations of our approach.

```
construct [{cplist}]"for each"{ var1 } "in"{stream1}","
                                { var2 } "in"{stream2 }"do"
                                { body }
                          "return"{ explist }
        ≡ (I',J' ← (for each i in {stream1} do
                      I',J' ← (for each j in {stream2 }do
                                 return all i, all j)
                    return all I', all J')
          return (initial {cplist}← {cplist}
                  for each{ var1 } in I';{var2}in J' do
                    new ( { cplist} ) ←{body }
                  return{ explist} ))                    (5.12)

[x,y,I,J ]"for each" i "in" I "," j "in" J "do"
                     f(x,i,j), f(y,i,j)
          "return" x,y
using [xo,yo,I,J ]                                      (5.13)
```

Since the values of the streams I and J should be supplied from outside, they occur in the 'list' in expression (5.13). At the time of substitution, { cplist} ≡ x,y,I,J and thus statement new ({ cplist })← { body }will be translated as

new x, new y, new I, new J ← { body }.

This will result in circulation of streams I and J in the

second loop of expression (5.12). Circulation of streams is to be avoided for the obvious reasons of efficiency. In fact in this case second loop never uses the streams I and J. An optimising compiler may detect this fact and treat I and J as dead variables.

A way to avoid such a circulation of streams would be to define a deletion operation on { cplist }. If we denote the deletion operation by - then we can write

new ({cplist} - {stream1 } - {stream2 }) ←{ body }

which, when translated at the substitution time, would result in

new x, new y ← { body}.

Then one can also think of defining a complementary operation addition on { cplist } .

5.4 Limitations of our approach

We pointed out a limitation of our approach in example (d) in Section 5.3 and suggested a solution. The limitation arose from our need to distinguish between circulating and uncirculating variables in a loop expression for the reasons of efficiency. Sometimes a programmer may want to circulate a stream to obtain desired results. Our proposed solution is neither elegant nor general enough.

It also increases the complexity of 'subst' in the implementation. A general solution to distinguish circulating variables from uncirculating variables will require a complete analysis of the program.

Second limitation or restriction is pointed out in example (a) in Section 5.3, i.e. a recursively defined pdc should return values for all the variables in 'list'.

Finally we would like to point out that our approach is not dynamic enough. Consider the definition of a case expression as given in expression (5.2). This expression can deal with only two cases. We have to define separate case expressions to deal with three cases, four cases and so on. A general case expression dealing with any number of cases cannot be defined in our model because a pdc is ultimately converted to a procedure and thus cannot model dynamicness in its definition itself. Conversion of a pdc to a procedure also reduces the interaction between the use of the pdc and the program where it is being used.

5.5 Summary

In this chapter we have discussed the definition, the use, and the implementation of programmer defined constructs in Id. Programmer defined constructs are useful in abstraction

of operations and in defining one's own language. The definition of a pdc specifies the relationship among expressions in terms of Id schemas and dummy expressions. The actual expressions are supplied by the use of a pdc. To bring out certain important points and limitations of our approach we have discussed several examples.

CHAPTER 6

CONCLUSIONS

In this thesis we have proposed certain extensions to the high level data flow language Id so that its basic mechanisms can be fully exploited. It is evident from the discussions in Chapter 3 and 4 that some of these extensions, can be incorporated without any major alteration in the set of base language operators. An alternate translation of the for loop and the for each-while loop as discussed in Chapter 3, enables us to incorporate sequential and mixed looping constructs and to enumerate the elements of a pdt in a controlled fashion. Similarly in Chapter 4 a slight modification of monitors enabled us to exploit the stream structures in the resource management problems. This points to the fact that the basic mechanisms in Id are powerful enough and their full potential is yet to be realized . In Chapter 5 programmer defined constructs were proposed as an addition to the basic mechanisms in Id. In fact the pdcs can be used to implement most of the constructs discussed in Chapter 3 and thus effectively reduce the size of the language.

Many ideas from Id and this thesis can also be extended to the conventional languages. The structure of the looping constructs in Id explicitly specifies the initialization

of the loop variables, and, an Id loop can terminate only when the associated predicate turns false. Such a structure of loops in a conventional language will facilitate program verification. An equivalent of parallel for each-while loop construct alongwith the generators will be able to model nearly all looping constructs in a conventional programming language. Recent languages such as ALPHARD have adopted such a view of loop constructs.

Another idea which can be extended to conventional programming languages is that of programmer defined constructs type. Using the conventions of Chapter 5, below we briefly outline how pdcs can be incorporated in the conventional languages:

Whenever a construct definition is encountered, it is entered in a structure, say environment. The user (always) specifies the name of the environment he will be using during the execution of his program. Initially environment may be an empty structure. Whenever a construct use is encountered it is translated into two statements:

1. a call to a system procedure evalc with the name of the construct and a structure containing actual expressions,
2. a call to the procedure, whose name is the first construct word, with the arguments specified in the argument list of the construct use.

The system procedure evalc substitutes the actual expressions in the construct definition, creates a procedure from the substituted definition, and then calls the compiler to translate the created procedure. A dynamic linking and loading facility will make the procedure available to the program. For incorporation of the mechanism in the languages which do not view the procedures as functions, the {cplist} needs to be extended to include the variables whose values are being returned by the construct. The arguments are transferred using 'call by value'.

During the course of this work we recognized many areas which are still unexplored or partially explored. Our discussion on pdcs is certainly not the last word on the subject. There are certain limitations in the proposed approach and it would be worthwhile to investigate how these limitations can be removed from the present frame work. Another area of interest is the specification of a minimal set of Id constructs which alongwith the facility of pdcs are powerful enough to model any programming problem. In specifying such a minimal set, the efficiency consideration should not be unduly compromised. Also the constructs in the minimal set should not be at such a primitive level that one has to deal with the base language.

Recent Turing lecture by J. Backus [Bac78] has brought functional forms of programming languages into vogue. At this stage we speculate that if the unravelling interpreter can be made to execute functional programs efficiently, then the resulting system will be able to do away with most of the objections against conventional systems (both software and hardware) and can emerge as a powerful next generation system.

REFERENCES

[AG77a] Arvind, and Gostelow, K.P. Some relationship between asynchronous interpreters of a data flow language. In Formal Description of Programming Languages, E.J. Neuhold, Ed., North-Holland Pub. Co., New York, 1978.

[AG77b] Arvind, and Gostelow, K.P. A computer capable of exchanging processing elements for time. In Information Processing 77, B. Gilchrist, Ed., North-Holland Pub. Co., New York, 1977.

[AG77c] Arvind, Gostelow, K.P. Microelectronics and Computer Science. Proceedings of the 2nd IEEE (G PHP)/ISHM University/Industry/Government Microelectronics Symposium, University of New Mexico, Albuquerque, New Mexico, January,1977.

[AG78] Arvind, and Gostelow, K.P. Data flow computer architecture: research and goals. Technical Report # 113, Deptt. of Information and Computer Science, University of California, Irvine, CA, Feb.,1978.

[AGP77] Arvind, Gostelow, K.P., and Plouffe, W.E. Indeterminacy, monitors, and dataflow. Proc. Sixth ACM Symp. on Operating Systems Principles, Nov.,1977, pp. 159-169.

[AGP78] Arvind, Gostelow, K.P., and Plouffe, W.E. The (preliminary) Id report: an asynchronous programming language and computing machine. Technical Report # 114, Deptt. of Information and Computer Science, University of California, Irvine, CA, May,1978.

[AW77] Ashcroft,E.A., and Wadge, W.W. LUCID, a nonprocedural language with iteration. Comm. ACM 20, 7(July,1977), 519-526.

[Bac73] Backus, J. Programming language semantics and closed applicative languages. Rep. RJ 1245, IBM Thomas J. Watson Research Centre, Yorktown Heights, N.Y., July,1973.

[Bac78] Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Rep. RJ 2234, IBM Research Lab., San Jose, California, April,1978.

[Bah72] Bahrs, A. Operations patterns. Symposium on Theoretical Programming, Novosibirsk, USSR, August,1972, pp. 217-246.

[Bri73] Brinch Hansen, P. Operating Systems Principles. Prentice Hall, Englewood Cliffs, N.J., 1973.

[Bry77] Bryant, R.E. Parallel programming. Computation Structures Group Memo 148, Lab. for Computer Science, Cambridge, MA, July,1977.

[Den73] Dennis, J.B. First version of a data flow procedure language. Computation Structures Group Memo 93, Lab. for Computer Science, Cambridge, MA, Nov.,1973. (revised as MAC Technical Memorandum 61, May,1975).

[Dij68] Dijkstra, E.W. Co-operating sequential processes. Programming Languages, F. Genuys, Ed., Academic Press, N.Y.,1968.

[GIMT74] Glushkov, V.M., Ignatyev, M.B., Myasnikov, V.A., and Torgashev, V.A. Recursive machines and computing technology. In Information Processing 74, North-Holland Pub. Co., N.Y.,1974.

[Gut77] Guttag, J. Abstract data types and the development of data structures. Comm. ACM 20,6 (June,1977), 396-404.

[HOS78] Hankin, C.L., Osmon, P.E., and Sharp, J.A. A data flow model of computation. Deptt. of Computer Science, Westfield College, Hampstead, London, 1978.

[Hoa74] Hoare, C.A.R. Monitors: An operating system structuring concept. Comm. ACM 17, 10 (Oct.,1974), 549-557.

[JL76] Jones, A.V., and Liskov, B.H. An access control facility for programming languages. Deptt. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, PA, May,1976.

[Kat78a] Kathail, V.K. Merge-sort procedure in Id. I.I.T. Data flow Note # 1, Computer Science Programme, Indian Institute of Technology, Kanpur, India, Feb.,1978.

[Kat78b] Kathail, V.K. Fast Fourier Transform in Id. I.I.T. Data flow Note # 2, Computer Science Programme, Indian Institute of Technology, Kanpur, India, March,1978.

[KM66] Karp, R.M.,and Miller, R.E. Properties of a model for parallel computations: determinacy, termination and queuing. SIAM J. Appl. Math. 11, 6 (Nov.,1966).

[Kos73] Kosinski, P.R. A data flow language for operating systems programming, ACM SIGPLAN Notices 8, 9 (Sept.,1973), 89-94.

[LHLMP77] Lampson, B., Horning, J., London, R., Mitchell, J., and Popek, G. Report on the programming language EUCLID. ACM SIGPLAN Notices 12, 2 (Feb.,1977), 1-79.

[LSAS77] Liskov, B., Syndar, A., Atkinson, R., and Schaffert, C. Abstraction mechanisms in CLU. Comm. ACM 20, 8 (Aug.,1977), 564-576.

[Mal78] Malhotra, V.M. Algorithms and data flow languages. I.I.T. Data flow Note # 3, Computer Science Programme, Indian Institute of Technology, Kanpur, India, June,1978.

[McC62] McCarthy, J. etal. LISP 1.5 Programmer's Manual. M.I.T. Press, Cambridge, Mass., 1962.

[Pra78] Pratt, T.W. Control computation and the design of loop control structures. IEEE Transactions on Software Engineering SE-4, 2 (March,1978), 81-89.

[Rus78] Russel, R.M. The CRAY-1 computer system. Comm. ACM 21, 1 (Jan.,1977), 63-72.

[Sha74] Shaw, A.C. The Logical Design of Operating Systems. Prentice Hall, Englewood Cliffs, N.J., 1974.

[SM77] Sutherland, I.E., and Mead, C.A. Microelectronics and Computer Science. Scientific American 237, 3 (Sept.,1977), 210-228.

[Str65] Strachey, C. A general purpose macrogenerator. Computer Journal 8, 3 (1965),225-241.

[SWL77] Shaw, M., Wulf, W.A., and London, R.L. Abstraction and verification in Alphard: defining and specifying iteration and generators. Comm. ACM 20, 8 (Aug.,1977), 533-564.

[Wen75] Weng, K.S. Stream-oriented computation in recursive data flow schemas. MAC Technical Memorandum 68, Lab. for Computer Science, Cambridge, MA, Oct.,1975.

[WLS76] Wulf, W.A., London, R.L., and Shaw, M. An introduction to the construction and verification of Alphard programs. IEEE Transactions on Software Engineering SE-2, 4 (Dec.,1976), 253-265.